目录

目录

附录

PHP7内核剖析

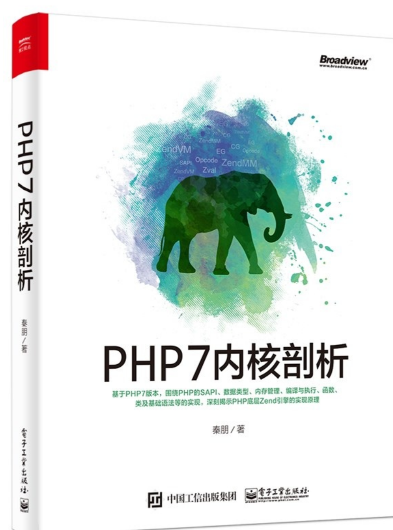
```
原创内容，转载请注明出处~

代码版本：php-7.0.12
```

反馈

交流&吐槽 错误反馈

纸质版



(coming soon)

目录：

附录

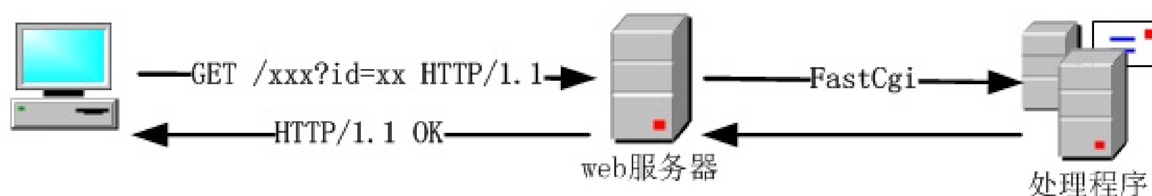
1.3 FPM

1.3.1 概述

FPM(FastCGI Process Manager)是PHP FastCGI运行模式的一个进程管理器，从它的定义可以看出，FPM的核心功能是进程管理，那么它用来管理什么进程呢？这个问题就需要从FastCGI说起了。

FastCGI是Web服务器(如：Nginx、Apache)和处理程序之间的一种通信协议，它是与Http类似的一种应用层通信协议，注意：它只是一种协议！

前面曾一再强调，PHP只是一个脚本解析器，你可以把它理解为一个普通的函数，输入是PHP脚本。输出是执行结果，假如我们想用PHP代替shell，在命令行中执行一个文件，那么就可以写一个程序来嵌入PHP解析器，这就是cli模式，这种模式下PHP就是普通的一个命令工具。接着我们又想：能不能让PHP处理http请求呢？这时就涉及到了网络处理，PHP需要接收请求、解析协议，然后处理完成返回请求。在网络应用场景下，PHP并没有像Golang那样实现http网络库，而是实现了FastCGI协议，然后与web服务器配合实现了http的处理，web服务器来处理http请求，然后将解析的结果再通过FastCGI协议转发给处理程序，处理程序处理完成后将结果返回给web服务器，web服务器再返回给用户，如下图所示。



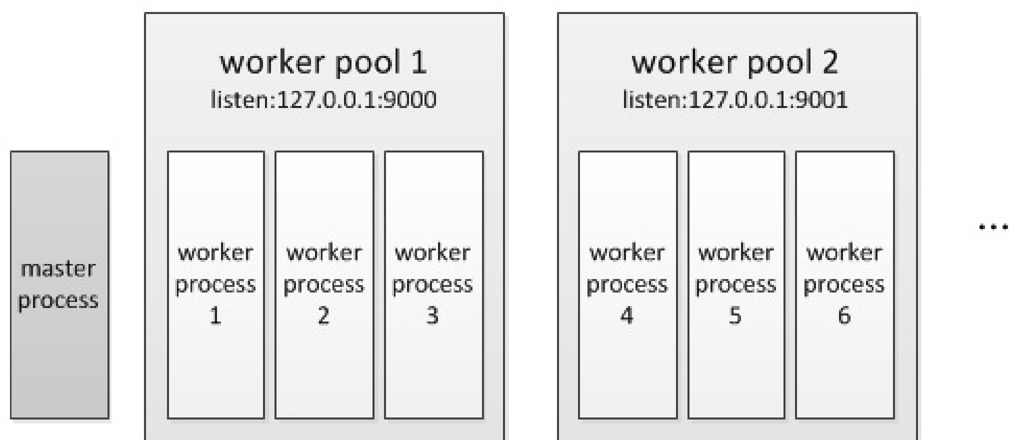
PHP实现了FastCGI协议的解析，但是并没有具体实现网络处理，一般的处理模型：多进程、多线程，多进程模型通常是主进程只负责管理子进程，而基本的网络事件由各个子进程处理，nginx、fpm就是这种模式；另一种多线程模型与多进程类似，只是它是线程粒度，通常会由主线程监听、接收请求，然后交由子线程处理，memcached就是这种模式，有的也是采用多进程那种模式：主线程只负责管理子线程不处理网络事件，各个子线程监听、接收、处理请求，memcached使用udp协议时采用的是这种模式。

1.3.2 基本实现

概括来说，fpm的实现就是创建一个master进程，在master进程中创建并监听socket，然后fork出多个子进程，这些子进程各自accept请求，子进程的处理非常简单，它在启动后阻塞在accept上，有请求到达后开始读取请求数据，读取完成后开始处理然后再返回，在这期间是不会接收其它请求的，也就是说fpm的子进程同时只能响应一个请求，只有把这个请求处理完成后才会accept下一个请求，这一点与nginx的事件驱动有很大的区别，nginx的子进程通过epoll管理套接字，如果一个请求数据还未发送完成则会处理下一个请求，即一个进程会同时连接多个请求，它是非阻塞的模型，只处理活跃的套接字。

fpm的master进程与worker进程之间不会直接进行通信，master通过共享内存获取worker进程的信息，比如worker进程当前状态、已处理请求数等，当master进程要杀掉一个worker进程时则通过发送信号的方式通知worker进程。

fpm可以同时监听多个端口，每个端口对应一个worker pool，而每个pool下对应多个worker进程，类似nginx中server概念。



在php-fpm.conf中通过 `[pool name]` 声明一个worker pool：

```
[web1]
listen = 127.0.0.1:9000
...

[web2]
listen = 127.0.0.1:9001
...
```

启动fpm后查看进程：ps -aux|grep fpm

```
root      27155  0.0  0.1 144704  2720 ?        Ss   15:16   0:00
 php-fpm: master process (/usr/local/php7/etc/php-fpm.conf)
nobody    27156  0.0  0.1 144676  2416 ?        S    15:16   0:00
 php-fpm: pool web1
nobody    27157  0.0  0.1 144676  2416 ?        S    15:16   0:00
 php-fpm: pool web1
nobody    27159  0.0  0.1 144680  2376 ?        S    15:16   0:00
 php-fpm: pool web2
nobody    27160  0.0  0.1 144680  2376 ?        S    15:16   0:00
 php-fpm: pool web2
```

具体实现上worker pool通过 `fpm_worker_pool_s` 这个结构表示，多个worker pool组成一个单链表：

```
struct fpm_worker_pool_s {
    struct fpm_worker_pool_s *next; //指向下一个worker pool
    struct fpm_worker_pool_config_s *config; //conf配置:pm、max_children、start_servers...
    int listening_socket; //监听的套接字
    ...

    //以下这个值用于master定时检查、记录worker数
    struct fpm_child_s *children; //当前pool的worker链表
    int running_children; //当前pool的worker运行总数
    int idle_spawn_rate;
    int warn_max_children;

    struct fpm_scoreboard_s *scoreboard; //记录worker的运行信息，比如空闲、忙碌worker数
    ...
}
```

1.3.3 FPM的初始化

接下来看下fpm的启动流程，从 `main()` 函数开始：

```
//sapi/fpm/fpm/fpm_main.c
int main(int argc, char *argv[])
{
    ...
    //注册SAPI:将全局变量sapi_module设置为cgi_sapi_module
    sapi_startup(&cgi_sapi_module);
    ...
    //执行php_module_starup()
    if (cgi_sapi_module.startup(&cgi_sapi_module) == FAILURE) {
        return FPM_EXIT_SOFTWARE;
    }
    ...
    //初始化
    if(0 > fpm_init(...)){
        ...
    }
    ...
    fpm_is_running = 1;

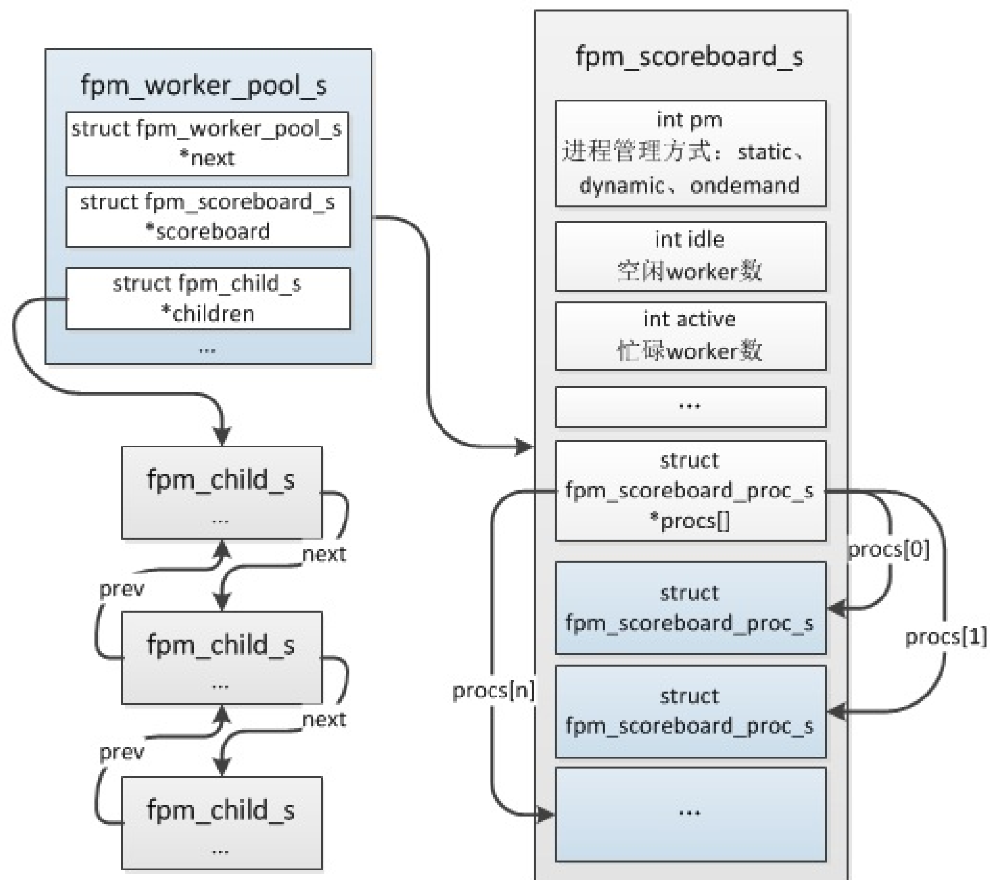
    fcgi_fd = fpm_run(&max_requests);//后面都是worker进程的操作，ma
ster进程不会走到下面
    parent = 0;
    ...
}
```

`fpm_init()` 主要有以下几个关键操作：

(1)fpm_conf_init_main():

解析php-fpm.conf配置文件，分配worker pool内存结构并保存到全局变量中：fpm_worker_all_pools，各worker pool配置解析到 `fpm_worker_pool_s->config` 中。

(2)fpm_scoreboard_init_main(): 分配用于记录worker进程运行信息的共享内存，按照worker pool的最大worker进程数分配，每个worker pool分配一个 `fpm_scoreboard_s` 结构，pool下对应的每个worker进程分配一个 `fpm_scoreboard_proc_s` 结构，各结构的对应关系如下图。



(3)fpm_signals_init_main():

```
static int sp[2];

int fpm_signals_init_main()
{
    struct sigaction act;

    //创建一个全双工管道
    if (0 > socketpair(AF_UNIX, SOCK_STREAM, 0, sp)) {
        return -1;
    }
    //注册信号处理handler
    act.sa_handler = sig_handler;
    sigfillset(&act.sa_mask);
    if (0 > sigaction(SIGTERM,  &act, 0) ||
        0 > sigaction(SIGINT,   &act, 0) ||
        0 > sigaction(SIGUSR1,  &act, 0) ||
        0 > sigaction(SIGUSR2,  &act, 0) ||
        0 > sigaction(SIGCHLD,  &act, 0) ||
        0 > sigaction(SIGQUIT,  &act, 0)) {
        return -1;
    }
    return 0;
}
```

这里会通过 `socketpair()` 创建一个管道，这个管道并不是用于master与worker进程通信的，它只在master进程中使用，具体用途在稍后介绍event事件处理时再作说明。另外设置master的信号处理handler，当master收到SIGTERM、SIGINT、SIGUSR1、SIGUSR2、SIGCHLD、SIGQUIT这些信号时将调用 `sig_handler()` 处理：

```
static void sig_handler(int signo)
{
    static const char sig_chars[NSIG + 1] = {
        [SIGTERM] = 'T',
        [SIGINT]  = 'I',
        [SIGUSR1] = '1',
        [SIGUSR2] = '2',
        [SIGQUIT] = 'Q',
        [SIGCHLD] = 'C'
    };
    char s;
    ...
    s = sig_chars[signo];
    //将信号通知写入管道sp[1]端
    write(sp[1], &s, sizeof(s));
    ...
}
```

(4)fpm_sockets_init_main()

创建每个worker pool的socket套接字。

(5)fpm_event_init_main():

启动master的事件管理，fpm实现了一个事件管理器用于管理IO、定时事件，其中IO事件通过kqueue、epoll、poll、select等管理，定时事件就是定时器，一定时间后触发某个事件。

在 `fpm_init()` 初始化完成后接下来就是最关键的 `fpm_run()` 操作了，此环节将fork子进程，启动进程管理器，另外master进程将不会再返回，只有各worker进程会返回，也就是说 `fpm_run()` 之后的操作均是worker进程的。

```
int fpm_run(int *max_requests)
{
    struct fpm_worker_pool_s *wp;
    for (wp = fpm_worker_all_pools; wp; wp = wp->next) {
        //调用fpm_children_make() fork子进程
        is_parent = fpm_children_create_initial(wp);

        if (!is_parent) {
            goto run_child;
        }
    }
    //master进程将进入event循环，不再往下走
    fpm_event_loop(0);

run_child: //只有worker进程会到这里

    *max_requests = fpm_globals.max_requests;
    return fpm_globals.listening_socket; //返回监听的套接字
}
```

在fork后worker进程返回了监听的套接字继续main()后面的处理，而master将永远阻塞在 `fpm_event_loop()` ，接下来分别介绍master、worker进程的后续操作。

1.3.4 请求处理

`fpm_run()` 执行后将fork出worker进程，worker进程返回 `main()` 中继续向下执行，后面的流程就是worker进程不断accept请求，然后执行PHP脚本并返回。整体流程如下：

- **(1)**等待请求： worker进程阻塞在fcgi_accept_request()等待请求；
- **(2)**解析请求： fastcgi请求到达后被worker接收，然后开始接收并解析请求数据，直到request数据完全到达；
- **(3)**请求初始化： 执行php_request_startup()，此阶段会调用每个扩展的：PHP_RINIT_FUNCTION()；
- **(4)**编译、执行： 由php_execute_script()完成PHP脚本的编译、执行；
- **(5)**关闭请求： 请求完成后执行php_request_shutdown()，此阶段会调用每个扩展的：PHP_RSHUTDOWN_FUNCTION()，然后进入步骤(1)等待下一个请

求。

```
int main(int argc, char *argv[])
{
    ...
    fcgi_fd = fpm_run(&max_requests);
    parent = 0;

    //初始化fastcgi请求
    request = fpm_init_request(fcgi_fd);

    //worker进程将阻塞在这，等待请求
    while (EXPECTED(fcgi_accept_request(request) >= 0)) {
        SG(server_context) = (void *) request;
        init_request_info();

        //请求开始
        if (UNEXPECTED(php_request_startup() == FAILURE)) {
            ...
        }
        ...

        fpm_request_executing();
        //编译、执行PHP脚本
        php_execute_script(&file_handle);
        ...
        //请求结束
        php_request_shutdown((void *) 0);
        ...
    }
    ...
    //worker进程退出
    php_module_shutdown();
    ...
}
```

worker进程一次请求的处理被划分为5个阶段：

- **FPM_REQUEST_ACCEPTING:** 等待请求阶段

- **FPM_REQUEST_READING_HEADERS:** 读取fastcgi请求header阶段
- **FPM_REQUEST_INFO:** 获取请求信息阶段，此阶段是将请求的method、query stirng、request uri等信息保存到各worker进程的fpm_scoreboard_proc_s结构中，此操作需要加锁，因为master进程也会操作此结构
- **FPM_REQUEST_EXECUTING:** 执行请求阶段
- **FPM_REQUEST_END:** 没有使用
- **FPM_REQUEST_FINISHED:** 请求处理完成

worker处理到各个阶段时将会把当前阶段更新到 `fpm_scoreboard_proc_s->request_stage` ，master进程正是通过这个标识判断worker进程是否空闲的。

1.3.5 进程管理

这一节我们来看下master是如何管理worker进程的，首先介绍下三种不同的进程管理方式：

- **static:** 这种方式比较简单，在启动时master按照 `pm.max_children` 配置fork出相应数量的worker进程，即worker进程数是固定不变的
- **dynamic:** 动态进程管理，首先在fpm启动时按照 `pm.start_servers` 初始化一定数量的worker，运行期间如果master发现空闲worker数低于 `pm.min_spare_servers` 配置数(表示请求比较多，worker处理不过来了)则会fork worker进程，但总的worker数不能超过 `pm.max_children` ，如果master发现空闲worker数超过了 `pm.max_spare_servers` (表示闲着的worker太多了)则会杀掉一些worker，避免占用过多资源，master通过这4个值来控制worker数
- **ondemand:** 这种方式一般很少用，在启动时不分配worker进程，等到有请求了后再通知master进程fork worker进程，总的worker数不超过 `pm.max_children` ，处理完成后worker进程不会立即退出，当空闲时间超过 `pm.process_idle_timeout` 后再退出

前面介绍到在 `fpm_run()` master进程将进入 `fpm_event_loop()` ：

```
void fpm_event_loop(int err)
{
    //创建一个io read的监听事件，这里监听的就是在fpm_init()阶段中通过soc
ketpair()创建管道sp[0]
    //当sp[0]可读时将回调fpm_got_signal()
    fpm_event_set(&signal_fd_event, fpm_signals_get_fd(), FPM_EV
_READ, &fpm_got_signal, NULL);
    fpm_event_add(&signal_fd_event, 0);

    //如果在php-fpm.conf配置了request_terminate_timeout则启动心跳检查

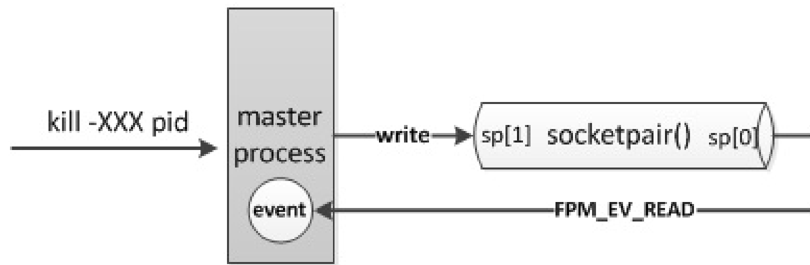
    if (fpm_globals.heartbeat > 0) {
        fpm_pctl_heartbeat(NULL, 0, NULL);
    }
    //定时触发进程管理
    fpm_pctl_perform_idle_server_maintenance_heartbeat(NULL, 0,
NULL);

    //进入事件循环，master进程将阻塞在此
    while (1) {
        ...
        //等待IO事件
        ret = module->wait(fpm_event_queue_fd, timeout);
        ...
        //检查定时器事件
        ...
    }
}
```

这就是master整体的处理，其进程管理主要依赖注册的几个事件，接下来我们详细分析下这几个事件的功能。

(1)sp[1]管道可读事件：

在 `fpm_init()` 阶段master曾创建了一个全双工的管道：sp，然后在这里创建了一个sp[0]可读的事件，当sp[0]可读时将交由 `fpm_got_signal()` 处理，向sp[1]写数据时sp[0]才会可读，那么什么时机会向sp[1]写数据呢？前面已经提到了：当master收到注册的那几种信号时会写入sp[1]端，这个时候将触发sp[0]可读事件。



这个事件是master用于处理信号的，我们根据master注册的信号逐个看下不同用途：

- **SIGINT/SIGTERM/SIGQUIT:** 退出fpm，在master收到退出信号后将向所有的worker进程发送退出信号，然后master退出
- **SIGUSR1:** 重新加载日志文件，生产环境中通常会对日志进行切割，切割后会生成一个新的日志文件，如果fpm不重新加载将无法继续写入日志，这个时候就需要向master发送一个USR1的信号
- **SIGUSR2:** 重启fpm，首先master也是会向所有的worker进程发送退出信号，然后master会调用execvp()重新启动fpm，最后旧的master退出
- **SIGCHLD:** 这个信号是子进程退出时操作系统发送给父进程的，子进程退出时，内核将子进程置为僵尸状态，这个进程称为僵尸进程，它只保留最小的一些内核数据结构，以便父进程查询子进程的退出状态，只有当父进程调用wait或者waitpid函数查询子进程退出状态后子进程才告终止，fpm中当worker进程因为异常原因(比如coredump了)退出而非master主动杀掉时master将受到此信号，这个时候父进程将调用waitpid()查下子进程的退出，然后检查下是不是需要重新fork新的worker

具体处理逻辑在 `fpm_got_signal()` 函数中，这里不再罗列。

(2)fpm_pctl_perform_idle_server_maintenance_heartbeat():

这是进程管理实现的主要事件，master启动了一个定时器，每隔1s触发一次，主要用于dynamic、ondemand模式下的worker管理，master会定时检查各worker pool的worker进程数，通过此定时器实现worker数量的控制，处理逻辑如下：

```
static void fpm_pctl_perform_idle_server_maintenance(struct timeval *now)
{
    for (wp = fpm_worker_all_pools; wp; wp = wp->next) {
        struct fpm_child_s *last_idle_child = NULL; //空闲时间最久的worker
```

```
        int idle = 0; //空闲worker数
        int active = 0; //忙碌worker数

        for (child = wp->children; child; child = child->next) {
            //根据worker进程的fpm_scoreboard_proc_s->request_stage判断
            if (fpm_request_is_idle(child)) {
                //找空闲时间最久的worker
                ...
                idle++;
            }else{
                active++;
            }
        }
        ...
        //ondemand模式
        if (wp->config->pm == PM_STYLE_ONDEMAND) {
            if (!last_idle_child) continue;

            fpm_request_last_activity(last_idle_child, &last);
            fpm_clock_get(&now);
            if (last.tv_sec < now.tv_sec - wp->config->pm_process_idle_timeout) {
                //如果空闲时间最长的worker空闲时间超过了process_idle_timeout则杀掉该worker
                last_idle_child->idle_kill = 1;
                fpm_pctl_kill(last_idle_child->pid, FPM_PCTL_QUIT);
            }
            continue;
        }
        //dynamic
        if (wp->config->pm != PM_STYLE_DYNAMIC) continue;
        if (idle > wp->config->pm_max_spare_servers && last_idle_child) {
            //空闲worker太多了，杀掉
            last_idle_child->idle_kill = 1;
            fpm_pctl_kill(last_idle_child->pid, FPM_PCTL_QUIT);
            wp->idle_spawn_rate = 1;
            continue;
```

```
        }
        if (idle < wp->config->pm_min_spare_servers) {
            //空闲worker太少了，如果总worker数未达到max数则fork
            ...
        }
    }
}
```

(3)fpm_pctl_heartbeat():

这个事件是用于限制worker处理单个请求最大耗时的，php-fpm.conf中有一个 `request_terminate_timeout` 的配置项，如果worker处理一个请求的总时长超过了这个值那么master将会向此worker进程发送 `kill -TERM` 信号杀掉worker进程，此配置单位为秒，默认值为0表示关闭此机制，另外fpm打印的slow log也是在这里完成的。

```
static void fpm_pctl_check_request_timeout(struct timeval *now)
{
    struct fpm_worker_pool_s *wp;

    for (wp = fpm_worker_all_pools; wp; wp = wp->next) {
        int terminate_timeout = wp->config->request_terminate_ti
meout;
        int slowlog_timeout = wp->config->request_slowlog_timeou
t;
        struct fpm_child_s *child;

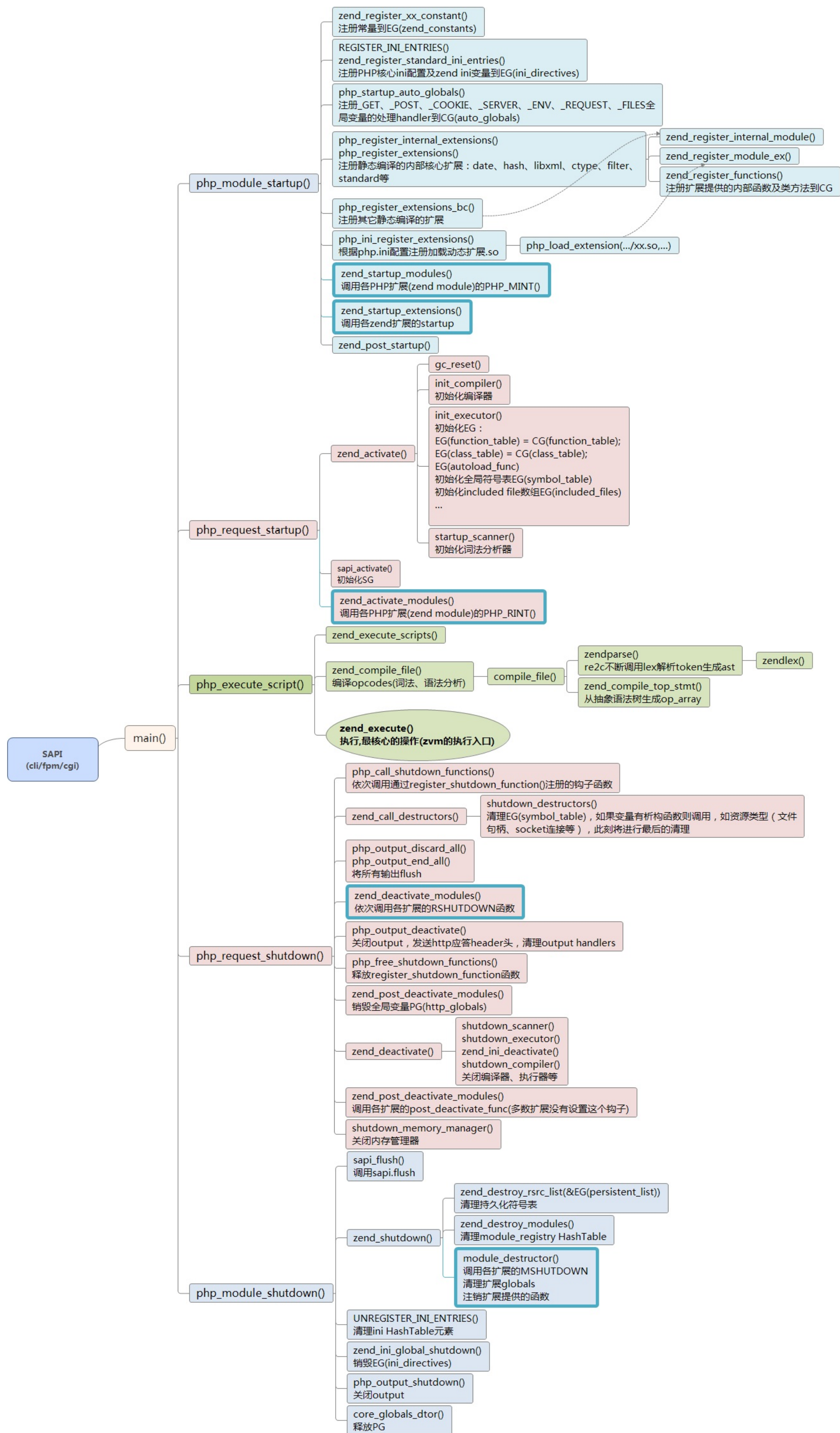
        if (terminate_timeout || slowlog_timeout) {
            for (child = wp->children; child; child = child->nex
t) {
                //检查当前当前worker处理的请求是否超时
                fpm_request_check_timed_out(child, now, terminat
e_timeout, slowlog_timeout);
            }
        }
    }
}
```

除了上面这几个事件外还有一个没有提到，那就是ondemand模式下master监听的新请求到达的事件，因为ondemand模式下fpm启动时是不会预创建worker的，有请求时才会生成子进程，所以请求到达时需要通知master进程，这个事件是在 `fpm_children_create_initial()` 时注册的，事件处理函数为 `fpm_pctl_on_socket_accept()` ，具体逻辑这里不再展开，比较容易理解。

到目前为止我们已经把fpm的核心实现介绍完了，事实上fpm的实现还是比较简单的。

1.2 执行流程

PHP的生命周期：



SAPI (cli/fpm/cgi)
main()
php_module_startup()
zend_register_xx_constant()
注册常量到EG(zend_constants)
REGISTER_INI_ENTRIES()
zend_register_standard_ini_entries()
注册PHP核心ini配置及zend ini变量到EG(ini_directives)
php_startup_auto_globals()
注册_GET、_POST、_COOKIE、_SERVER、_ENV、_REQUEST、_FILES全局变量的处理handler到CG(auto_globals)
php_register_internal_extensions()
php_register_extensions()
注册静态编译的内部核心扩展：date、hash、libxml、ctype、filter、standard等
zend_register_internal_module()
zend_register_module_ex()
zend_register_functions()
注册扩展提供的内部函数及类方法到CG
php_register_extensions_bc()
注册其它静态编译的扩展
php_ini_register_extensions()
根据php.ini配置注册加载动态扩展.so
php_load_extension(.../xx.so,...)
zend_startup_modules()
调用各PHP扩展(zend module)的PHP_MINT()
zend_startup_extensions()
调用各zend扩展的startup
zend_post_startup()
php_request_startup()
zend_activate()
gc_reset()
init_compiler()
初始化编译器
init_executor()
初始化EG：
EG(function_table) = CG(function_table);
EG(class_table) = CG(class_table);
EG(autoload_func)
初始化全局符号表EG(symbol_table)
初始化included file数组EG(included_files)
...
startup_scanner()
初始化词法分析器
sapi_activate()
初始化SG
zend_activate_modules()
调用各PHP扩展(zend module)的PHP_RINT()
php_execute_script()
zend_execute_scripts()
zend_compile_file()
编译opcodes(词法、语法分析)
compile_file()
zendparse()
re2c不断调用lex解析token生成ast
zendlex()
zend_compile_top_stmt()
从抽象语法树生成op_array
zend_execute()
执行,最核心的操作(zvm的执行入口)
php_request_shutdown()
php_call_shutdown_functions()
依次调用通过register_shutdown_function()注册的钩子函数
zend_call_destructors()
shutdown_destructors()
清理EG(symbol_table)，如果变量有析构函数则调用，如资源类型（文件句柄、socket连接等），此刻将进行最后的清理
php_output_discard_all()
php_output_end_all()
将所有输出flush
zend_deactivate_modules()
依次调用各扩展的RSHUTDOWN函数
php_output_deactivate()
关闭output，发送http应答header头，清理output handlers
php_free_shutdown_functions()
释放register_shutdown_function函数
zend_post_deactivate_modules()
销毁全局变量PG(http_globals)
zend_deactivate()
shutdown_scanner()
shutdown_executor()
zend_ini_deactivate()
shutdown_compiler()
关闭编译器、执行器等
zend_post_deactivate_modules()
调用各扩展的post_deactivate_func(多数扩展没有设置这个钩子)
shutdown_memory_manager()
关闭内存管理器
php_module_shutdown()
sapi_flush()
调用sapi.flush
zend_shutdown()
zend_destroy_rsrc_list(&EG(persistent_list))
清理持久化符号表
zend_destroy_modules()
清理module_registry HashTable
module_destructor()
调用各扩展的MSHUTDOWN
清理扩展globals
注销扩展提供的函数
UNREGISTER_INI_ENTRIES()
清理ini HashTable元素
zend_ini_global_shutdown()
销毁EG(ini_directives)
php_output_shutdown()
关闭output
core_globals_dtor()
释放PG

1.2.1 模块初始化阶段

1.2.2 请求初始化阶段

1.2.3 执行PHP脚本阶段

1.2.4 请求结束阶段

1.2.5 模块关闭阶段

2.1 变量的内部实现

变量是一个语言实现的基础，变量有两个组成部分：变量名、变量值，PHP中可以将其对应为：zval、zend_value，这两个概念一定要区分开，PHP中变量的内存是通过引用计数进行管理的，而且PHP7中引用计数是在zend_value而不是zval上，变量之间的传递、赋值通常也是针对zend_value。

PHP中可以通过 `$` 关键词定义一个变量： `$a;` ，在定义的同时可以进行初始化： `$a = "hi~";` ，注意这实际是两步：定义、初始化，只定义一个变量也是可以的，可以不给它赋值，比如：

```
$a;
$b = 1;
```

这段代码在执行时会分配两个zval。

接下来我们具体看下变量的结构以及不同类型的实现。

2.1.1 变量的基础结构

```
//zend_types.h
typedef struct _zval_struct     zval;

typedef union _zend_value {
    zend_long         lval;    //int整形
    double            dval;    //浮点型
    zend_refcounted  *counted;
    zend_string      *str;     //string字符串
    zend_array       *arr;     //array数组
    zend_object      *obj;     //object对象
    zend_resource    *res;     //resource资源类型
    zend_reference   *ref;     //引用类型，通过&$var_name定义的
    zend_ast_ref     *ast;     //下面几个都是内核使用的value
    zval             *zv;
    void             *ptr;
    zend_class_entry *ce;
```

```
    zend_function    *func;
    struct {
        uint32_t w1;
        uint32_t w2;
    } ww;
} zend_value;

struct _zval_struct {
    zend_value        value; //变量实际的value
    union {
        struct {
            ZEND_ENDIAN_LOHI_4( //这个是为了兼容大小字节序，小字节序就是下面的顺序，大字节序则下面4个顺序翻转
                zend_uchar    type,         //变量类型
                zend_uchar    type_flags,  //类型掩码，不同的类型会有不同的几种属性，内存管理会用到
                zend_uchar    const_flags,
                zend_uchar    reserved)     //call info，zend执行流程会用到
        } v;
        uint32_t type_info; //上面4个值的组合值，可以直接根据type_info取到4个对应位置的值
    } u1;
    union {
        uint32_t     var_flags;
        uint32_t     next;                 //哈希表中解决哈希冲突时用到
        uint32_t     cache_slot;           /* literal cache slot */
        uint32_t     lineno;               /* line number (for ast nodes) */
        uint32_t     num_args;             /* arguments number for EX(This) */
        uint32_t     fe_pos;               /* foreach position */

        uint32_t     fe_iter_idx;          /* foreach iterator index */
    } u2; //一些辅助值
};
```

`zval` 结构比较简单，内嵌一个union类型的 `zend_value` 保存具体变量类型的值或指针， `zval` 中还有两个union： `u1` 、 `u2` :

- **u1**： 它的意义比较直观，变量的类型就通过 `u1.v.type` 区分，另外一个值 `type_flags` 为类型掩码，在变量的内存管理、gc机制中会用到，第三部分会详细分析，至于后面两个 `const_flags` 、 `reserved` 暂且不管
- **u2**： 这个值纯粹是个辅助值，假如 `zval` 只有: `value` 、 `u1` 两个值，整个zval的大小也会对齐到16byte，既然不管有没有u2大小都是16byte，把多余的4byte拿出来用于一些特殊用途还是很划算的，比如next在哈希表解决哈希冲突时会用到，还有fe_pos在foreach会用到......

从 `zend_value` 可以看出，除 `long` 、 `double` 类型直接存储值外，其它类型都为指针，指向各自的结构。

2.1.2 类型

`zval.u1.type` 类型：

```
/* regular data types */
#define IS_UNDEF                    0
#define IS_NULL                     1
#define IS_FALSE                    2
#define IS_TRUE                     3
#define IS_LONG                     4
#define IS_DOUBLE                   5
#define IS_STRING                   6
#define IS_ARRAY                    7
#define IS_OBJECT                   8
#define IS_RESOURCE                 9
#define IS_REFERENCE                10

/* constant expressions */
#define IS_CONSTANT                 11
#define IS_CONSTANT_AST             12

/* fake types */
#define _IS_BOOL                    13
#define IS_CALLABLE                 14

/* internal types */
#define IS_INDIRECT                 15
#define IS_PTR                      17
```

2.1.2.1 标量类型

最简单的类型是true、false、long、double、null，其中true、false、null没有value，直接根据type区分，而long、double的值则直接存在value中：zend_long、double，也就是标量类型不需要额外的value指针。

2.1.2.2 字符串

PHP中字符串通过 `zend_string` 表示:

```
struct _zend_string {
    zend_refcounted_h gc;
    zend_ulong        h;                /* hash value */
    size_t            len;
    char              val[1];
};
```

- **gc**： 变量引用信息，比如当前value的引用数，所有用到引用计数的变量类型都会有这个结构，3.1节会详细分析
- **h**： 哈希值，数组中计算索引时会用到
- **len**： 字符串长度，通过这个值保证二进制安全
- **val**： 字符串内容，变长struct，分配时按len长度申请内存

事实上字符串又可具体分为几类：IS_STR_PERSISTENT(通过malloc分配的)、IS_STR_INTERNED(php代码里写的一些字面量，比如函数名、变量值)、IS_STR_PERMANENT(永久值，生命周期大于request)、IS_STR_CONSTANT(常量)、IS_STR_CONSTANT_UNQUALIFIED，这个信息通过flag保存：zval.value->gc.u.flags，后面用到的时候再具体分析。

2.1.2.3 数组

array是PHP中非常强大的一个数据结构，它的底层实现就是普通的有序HashTable，这里简单看下它的结构，下一节会单独分析数组的实现。

```
typedef struct _zend_array HashTable;

struct _zend_array {
    zend_refcounted_h gc; //引用计数信息，与字符串相同
    union {
        struct {
            ZEND_ENDIAN_LOHI_4(
                zend_uchar    flags,
                zend_uchar    nApplyCount,
                zend_uchar    nIteratorsCount,
                zend_uchar    reserve)
        } v;
        uint32_t flags;
    } u;
    uint32_t          nTableMask; //计算bucket索引时的掩码
    Bucket           *arData; //bucket数组
    uint32_t          nNumUsed; //已用bucket数
    uint32_t          nNumOfElements; //已有元素数，nNumOfElements
 <= nNumUsed，因为删除的并不是直接从arData中移除
    uint32_t          nTableSize; //数组的大小，为2^n
    uint32_t          nInternalPointer; //数值索引
    zend_long         nNextFreeElement;
    dtor_func_t       pDestructor;
};
```

2.1.2.4 对象/资源

```
struct _zend_object {
    zend_refcounted_h gc;
    uint32_t          handle;
    zend_class_entry *ce; //对象对应的class类
    const zend_object_handlers *handlers;
    HashTable        *properties; //对象属性哈希表
    zval              properties_table[1];
};

struct _zend_resource {
    zend_refcounted_h gc;
    int               handle;
    int               type;
    void             *ptr;
};
```

对象比较常见，资源指的是tcp连接、文件句柄等等类型，这种类型比较灵活，可以随意定义struct，通过ptr指向，后面会单独分析这种类型，这里不再多说。

2.1.2.5 引用

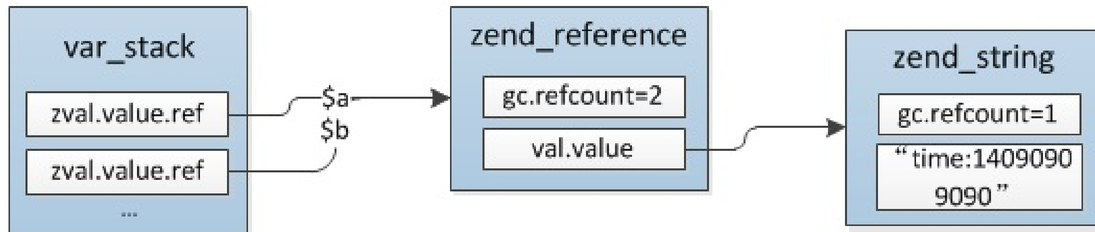
引用是PHP中比较特殊的一种类型，它实际是指向另外一个PHP变量，对它的修改会直接改动实际指向的zval，可以简单的理解为C中的指针，在PHP中通过 `&` 操作符产生一个引用变量，也就是说不管以前的类型是什么， `&` 首先会创建一个 `zend_reference` 结构，其内嵌了一个zval，这个zval的value指向原来zval的value(如果是布尔、整形、浮点则直接复制原来的值)，然后将原zval的类型修改为IS_REFERENCE，原zval的value指向新创建的 `zend_reference` 结构。

```
struct _zend_reference {
    zend_refcounted_h gc;
    zval              val;
};
```

结构非常简单，除了公共部分 `zend_refcounted_h` 外只有一个 `val` ，举个示例看下具体的结构关系：

```
$a = "time:" . time();      //$a     -> zend_string_1(refcount=1)
$b = &$a;                   //$a,$b -> zend_reference_1(refcount
=2) -> zend_string_1(refcount=1)
```

最终的结果如图：



注意：引用只能通过 `&` 产生，无法通过赋值传递，比如：

```
$a = "time:" . time();      //$a     -> zend_string_1(refcount=1)
$b = &$a;                   //$a,$b -> zend_reference_1(refcount
=2) -> zend_string_1(refcount=1)
$c = $b;                    //$a,$b -> zend_reference_1(refcount
=2) -> zend_string_1(refcount=2)
                            //$c     ->
       ---
```

`$b = &$a` 这时候 `$a` 、 `$b` 的类型是引用，但是 `$c = $b` 并不会直接将 `$b` 赋值给 `$c` ，而是把 `$b` 实际指向的zval赋值给 `$c` ，如果想要 `$c` 也是一个引用则需要这么操作：

```
$a = "time:" . time();      //$a        -> zend_string_1(refcount
=1)
$b = &$a;                   //$a,$b     -> zend_reference_1(refco
unt=2) -> zend_string_1(refcount=1)
$c = &$b;/*或$c = &$a*/     //$a,$b,$c -> zend_reference_1(refco
unt=3) -> zend_string_1(refcount=1)
```

这个也表示PHP中的 引用只可能有一层 ，不会出现一个引用指向另外一个引用的情况 ，也就是没有C语言中 `指针的指针` 的概念。

2.1.3 内存管理

接下来分析下变量的分配、销毁。

在分析变量内存管理之前我们先自己想一下可能的实现方案，最简单的处理方式：定义变量时alloc一个zval及对应的value结构(ref/arr/str/res...)，赋值、函数传参时硬拷贝一个副本，这样各变量最终的值完全都是独立的，不会出现多个变量同时共用一个value的情况，在执行完以后直接将各变量及value结构free掉。

这种方式是可行的，而且内存管理也很简单，但是，硬拷贝带来的一个问题是效率低，比如我们定义了一个变量然后赋值给另外一个变量，可能后面都只是只读操作，假如硬拷贝的话就会有多余的一份数据，这个问题的解决方案是：引用计数+写时复制 。PHP变量的管理正是基于这两点实现的。

2.1.3.1 引用计数

引用计数是指在value中增加一个字段 `refcount` 记录指向当前value的数量，变量复制、函数传参时并不直接硬拷贝一份value数据，而是将 `refcount++` ，变量销毁时将 `refcount--` ，等到 `refcount` 减为0时表示已经没有变量引用这个value，将它销毁即可。

```
$a = "time:" . time();   //$a       ->  zend_string_1(refcount=1)
$b = $a;                 //$a,$b    ->  zend_string_1(refcount=2)
$c = $b;                 //$a,$b,$c ->  zend_string_1(refcount=3)

unset($b);               //$b = IS_UNDEF  $a,$c ->  zend_string_1(refcount=2)
```

引用计数的信息位于给具体value结构的gc中：

```
typedef struct _zend_refcounted_h {
    uint32_t         refcount;          /* reference counter 32-
bit */
    union {
        struct {
            ZEND_ENDIAN_LOHI_3(
                zend_uchar    type,
                zend_uchar    flags,    /* used for strings & ob
jects */
                uint16_t      gc_info)  /* keeps GC root number
(or 0) and color */
        } v;
        uint32_t type_info;
    } u;
} zend_refcounted_h;
```

从上面的zend*value*结构可以看出并不是所有的数据类型都会用到引用计数，`long`、`double` 直接都是硬拷贝，只有*value*是指针的那几种类型才_可能会用到引用计数。

下面再看一个例子：

```
$a = "hi~";
$b = $a;
```

猜测一下变量 `$a/$b` 的引用情况。

这个不跟上面的例子一样吗？字符串 `"hi~"` 有 `$a/$b` 两个引用，所以 `zend_string1(refcount=2)` 。但是这是错的，gdb调试发现上面例子zend_string的引用计数为0。这是为什么呢？

```
$a,$b -> zend_string_1(refcount=0,val="hi~")
```

事实上并不是所有的PHP变量都会用到引用计数，标量：true/false/double/long/null是硬拷贝自然不需要这种机制，但是除了这几个还有两个特殊的类型也不会用到：interned string(内部字符串，就是上面提到的字符串flag：IS_STR_INTERNED)、immutable array，它们的type

是 `IS_STRING` 、 `IS_ARRAY` ，与普通string、array类型相同，那怎么区分一个value是否支持引用计数呢？还记得 `zval.u1` 中那个类型掩码 `type_flag` 吗？正是通过这个字段标识的，这个字段除了标识value是否支持引用计数外还有其它几个标识位，按位分割，注意： `type_flag` 与 `zval.value->gc.u.flag` 不是一个值。

支持引用计数的value类型其 `zval.u1.type_flag` 包含 (注意是&，不是等于) `IS_TYPE_REFCOUNTED` ：

```
#define IS_TYPE_REFCOUNTED          (1<<2)
```

下面具体列下哪些类型会有这个标识：

```
|     type       | refcounted |
+----------------+------------+
|simple types    |            |
|string          |      Y     |
|interned string |            |
|array           |      Y     |
|immutable array |            |
|object          |      Y     |
|resource        |      Y     |
|reference       |      Y     |
```

simple types很显然用不到，不再解释，string、array、object、resource、reference有引用计数机制也很容易理解，下面具体解释下另外两个特殊的类型：

- **interned string**： 内部字符串，这是种什么类型？我们在PHP中写的所有字符都可以认为是这种类型，比如function name、class name、variable name、静态字符串等等，我们这样定义: `$a = "hi~";` 后面的字符串内容是唯一不变的，这些字符串等同于C语言中定义在静态变量区的字符串： `char *a = "hi~";` ，这些字符串的生命周期为request期间，request完成后会统一销毁释放，自然也就无需在运行期间通过引用计数管理内存。
- **immutable array**： 只有在用opcache的时候才会用到这种类型，不清楚具体实现，暂时忽略。

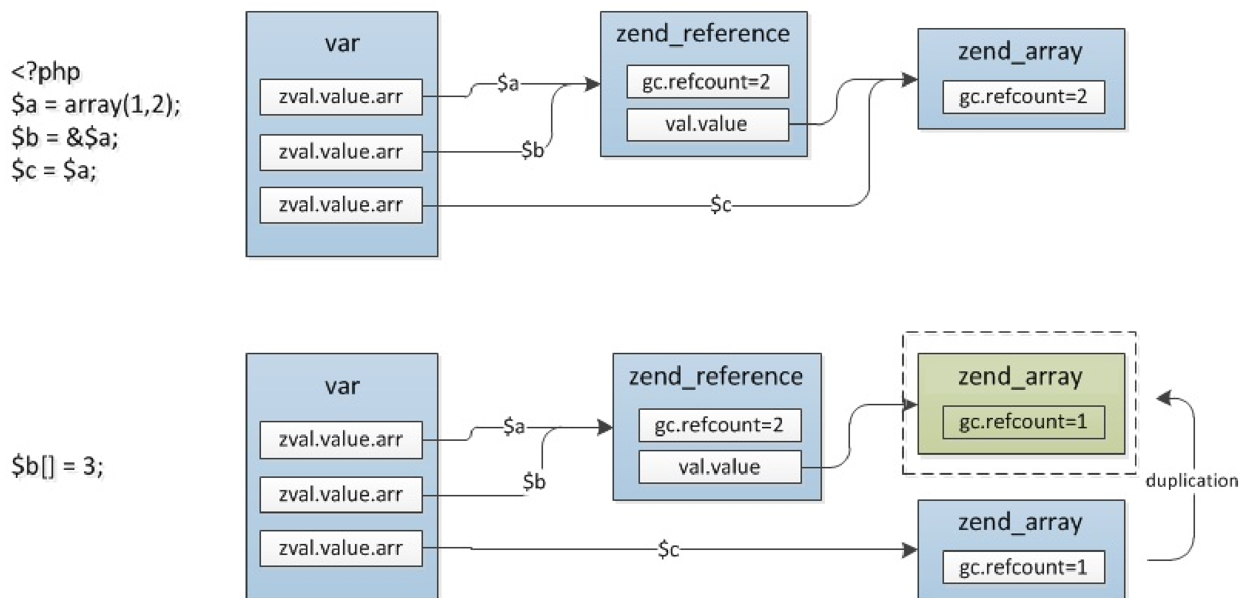
2.1.3.2 写时复制

上一小节介绍了引用计数，多个变量可能指向同一个value，然后通过refcount统计引用数，这时候如果其中一个变量试图更改value的内容则会重新拷贝一份value修改，同时断开旧的指向，写时复制的机制在计算机系统中有非常广的应用，它只有在必要的时候(写)才会发生硬拷贝，可以很好的提高效率，下面从示例看下：

```
$a = array(1,2);
$b = &$a;
$c = $a;

//发生分离
$b[] = 3;
```

最终的结果：



不是所有类型都可以copy的，比如对象、资源，实时上只有string、array两种支持，与引用计数相同，也是通过 `zval.u1.type_flag` 标识value是否可复制的：

```
#define IS_TYPE_COPYABLE         (1<<4)
```

```
|     type       |  copyable  |
+----------------+------------+
|simple types    |            |
|string          |      Y     |
|interned string |            |
|array           |      Y     |
|immutable array |            |
|object          |            |
|resource        |            |
|reference       |            |
```

copyable 的意思是当value发生duplication时是否需要或者能够copy，这个具体有两种情形下会发生：

- a.从 **literal**变量区 复制到 局部变量区 ，比如： `$a = [];` 实际会有两个数组，而 `$a = "hi~";//interned string` 则只有一个string
- b.局部变量区分离时(写时复制)：如改变变量内容时引用计数大于1则需要分离， `$a = [];$b = $a; $b[] = 1;` 这里会分离，类型是array所以可以复制，如果是对象： `$a = new user;$b = $a;$a->name = "dd";` 这种情况是不会复制object的，$a、$b指向的对象还是同一个

具体literal、局部变量区变量的初始化、赋值后面编译、执行两篇文章会具体分析，这里知道变量有个 `copyable` 的属性就行了。

2.1.3.3 变量回收

PHP变量的回收主要有两种：主动销毁、自动销毁。主动销毁指的就是 **unset** ，而自动销毁就是PHP的自动管理机制，在return时减掉局部变量的refcount，即使没有显式的return，PHP也会自动给加上这个操作，另外一个就是写时复制时会断开原来value的指向，这时候也会检查断开后旧value的refcount。

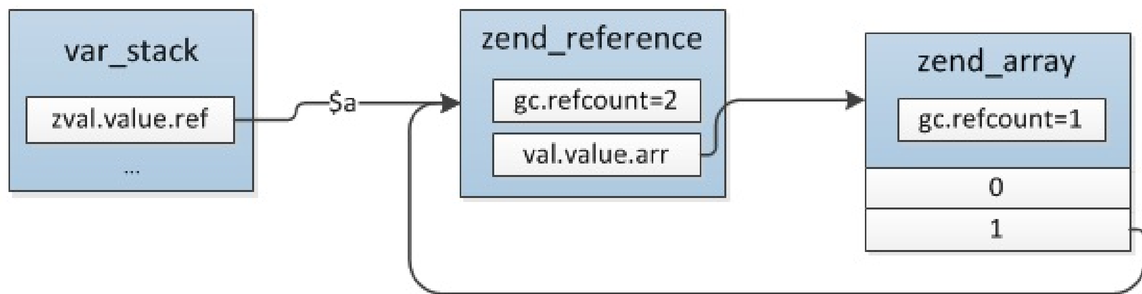
2.1.3.4 垃圾回收

PHP变量的回收是根据refcount实现的，当unset、return时会将变量的引用计数减掉，如果refcount减到0则直接释放value，这是变量的简单gc过程，但是实际过程中出现gc无法回收导致内存泄漏的bug，先看下一个例子：

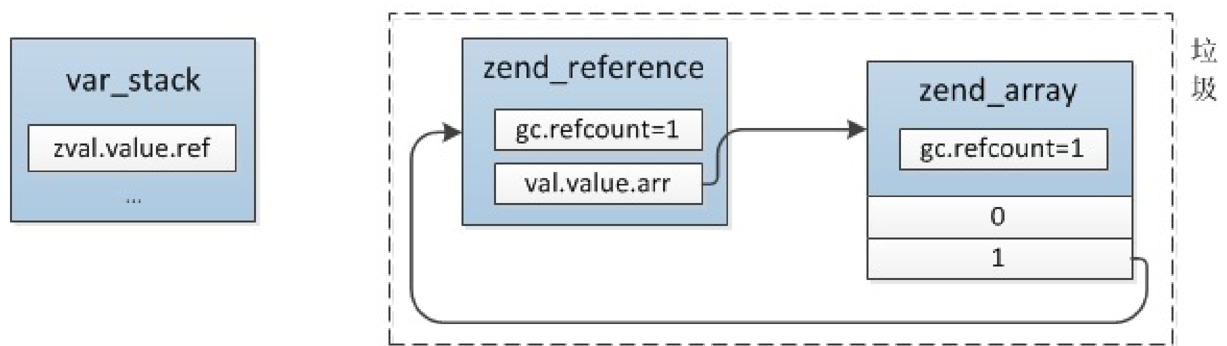
```
$a = [1];
$a[] = &$a;

unset($a);
```

`unset($a)` 之前引用关系：



`unset($a)` 之后：



可以看到， `unset($a)` 之后由于数组中有子元素指向 `$a` ，所以 `refcount > 0` ，无法通过简单的gc机制回收，这种变量就是垃圾，垃圾回收器要处理的就是这种情况，目前垃圾只会出现在array、object两种类型中，所以只会针对这两种情况作特殊处理：当销毁一个变量时，如果发现减掉refcount后仍然大于0，且类型是IS_ARRAY、IS_OBJECT则将此value放入gc可能垃圾双向链表中，等这个链表达到一定数量后启动检查程序将所有变量检查一遍，如果确定是垃圾则销毁释放。

标识变量是否需要回收也是通过 `u1.type_flag` 区分的：

```
#define IS_TYPE_COLLECTABLE
```

```
|     type        | collectable |
+----------------+-------------+
|simple types     |             |
|string           |             |
|interned string |             |
|array            |      Y      |
|immutable array |             |
|object           |      Y      |
|resource         |             |
|reference        |             |
```

具体的垃圾回收过程这里不再介绍，后面会单独分析。

2.2 数组

数组是PHP中非常强大、灵活的一种数据类型，它的底层实现为散列表(HashTable，也称作：哈希表)，除了我们熟悉的PHP用户空间的Array类型之外，内核中也随处用到散列表，比如函数、类、常量、已include文件的索引表、全局符号表等都用的HashTable存储。

散列表是根据关键码值(Key value)而直接进行访问的数据结构，它的key - value之间存在一个映射函数，可以根据key通过映射函数直接索引到对应的value值，它不以关键字的比较为基本操作，采用直接寻址技术（就是说，它是直接通过key映射到内存地址上去的），从而加快查找速度，在理想情况下，无须任何比较就可以找到待查关键字，查找的期望时间为O(1)。

2.2.1 数组结构

存放记录的数组称做散列表，这个数组用来存储value，而value具体在数组中的存储位置由映射函数根据key计算确定，映射函数可以采用取模的方式，key可以通过一些譬如“times 33”的算法得到一个整形值，然后与数组总大小取模得到在散列表中的存储位置。这是一个普通散列表的实现，PHP散列表的实现整体也是这个思路，只是有几个特殊的地方，下面就是PHP中HashTable的数据结构：

```
//Bucket：散列表中存储的元素
typedef struct _Bucket {
    zval              val; //存储的具体value，这里嵌入了一个zval，而不是一个指针
    zend_ulong        h;   //key根据times 33计算得到的哈希值，或者是数值索引编号
    zend_string      *key; //存储元素的key
} Bucket;

//HashTable结构
typedef struct _zend_array HashTable;
struct _zend_array {
    zend_refcounted_h gc;
    union {
        struct {
            ZEND_ENDIAN_LOHI_4(
                    zend_uchar    flags,
                    zend_uchar    nApplyCount,
                    zend_uchar    nIteratorsCount,
                    zend_uchar    reserve)
        } v;
        uint32_t flags;
    } u;
    uint32_t          nTableMask; //哈希值计算掩码，等于nTableSize的负值(nTableMask = -nTableSize)
    Bucket           *arData;     //存储元素数组，指向第一个Bucket
    uint32_t          nNumUsed;   //已用Bucket数
    uint32_t          nNumOfElements; //哈希表有效元素数
    uint32_t          nTableSize;     //哈希表总大小，为2的n次方
    uint32_t          nInternalPointer;
    zend_long         nNextFreeElement; //下一个可用的数值索引,如:arr[] = 1;arr["a"] = 2;arr[] = 3;  则nNextFreeElement = 2;
    dtor_func_t       pDestructor;
};
```

HashTable中有两个非常相近的值: `nNumUsed` 、 `nNumOfElements` ， `nNumOfElements` 表示哈希表已有元素数，那这个值不跟 `nNumUsed` 一样吗？为什么要定义两个呢？实际上它们有不同的含义，当将一个元素从哈希表删除时并不会将对应的Bucket移除，而是将Bucket存

储的zval修改为 `IS_UNDEF` ，只有扩容时发现nNumOfElements与nNumUsed相差达到一定数量(这个数量是: `ht->nNumUsed - ht->nNumOfElements > (ht->nNumOfElements >> 5)`)时才会将已删除的元素全部移除，重新构建哈希表。所以 `nNumUsed` >= `nNumOfElements` 。

HashTable中另外一个非常重要的值 `arData` ，这个值指向存储元素数组的第一个Bucket，插入元素时按顺序 依次插入 数组，比如第一个元素在arData[0]、第二个在arData[1]...arData[nNumUsed]。PHP数组的有序性正是通过 `arData` 保证的，这是第一个与普通散列表实现不同的地方。

既然arData并不是按key映射的散列表，那么映射函数是如何将key与arData中的value建立映射关系的呢？

实际上这个散列表也在 `arData` 中，比较特别的是散列表在ht->arData内存之前，分配内存时这个散列表与Bucket数组一起分配，arData向后移动到了Bucket数组的起始位置，并不是申请内存的起始位置，这样散列表可以由arData指针向前移动访问到，即arData[-1]、arData[-2]、arData[-3]......散列表的结构是 `uint32_t` ，它保存的是value在Bucket数组中的位置。

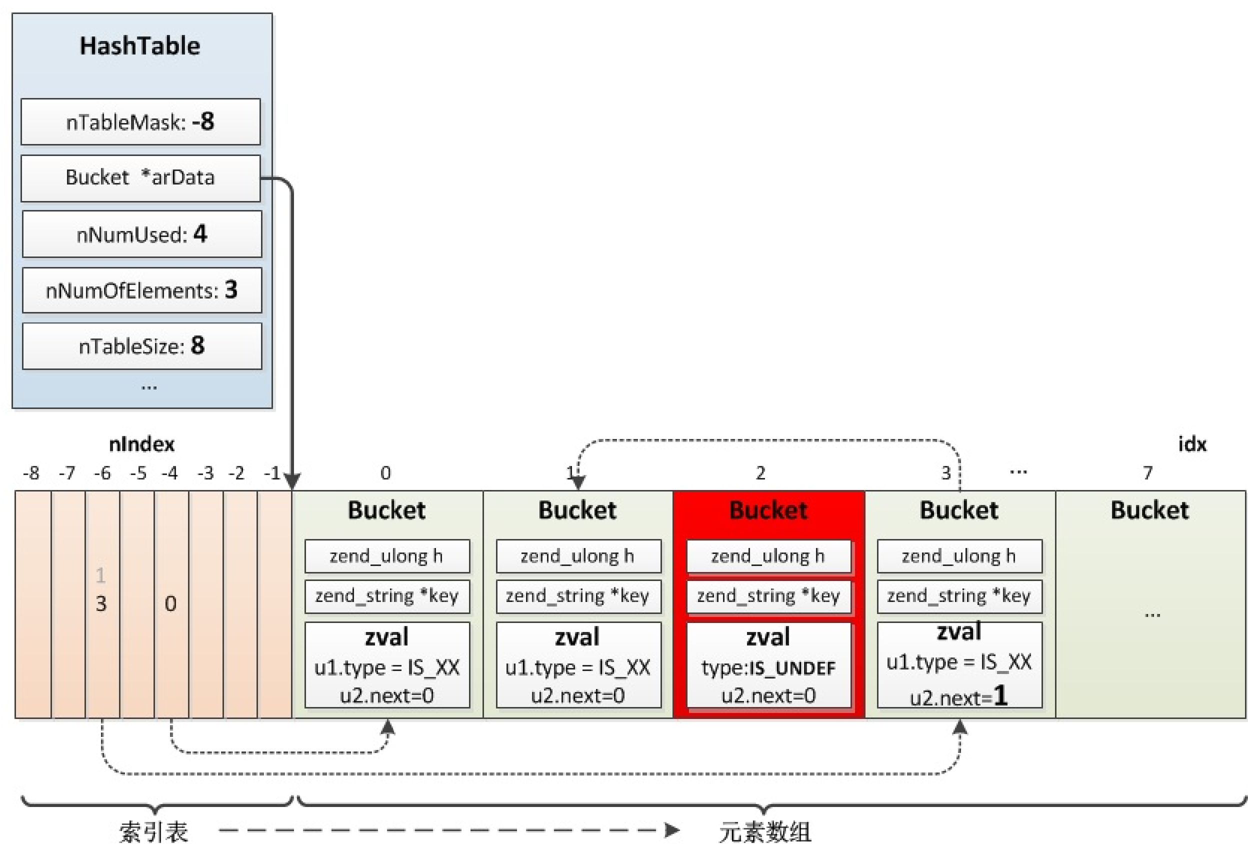
所以，整体来看HashTable主要依赖arData实现元素的存储、索引。插入一个元素时先将元素按先后顺序插入Bucket数组，位置是idx，再根据key的哈希值映射到散列表中的某个位置nIndex，将idx存入这个位置；查找时先在散列表中映射到nIndex，得到value在Bucket数组的位置idx，再从Bucket数组中取出元素。

比如：

```
$arr["a"] = 1;
$arr["b"] = 2;
$arr["c"] = 3;
$arr["d"] = 4;

unset($arr["c"]);
```

对应的HashTable如下图所示。



> 图中Bucket的zval.u2.next默认值应该为-1，不是0

2.2.2 映射函数

映射函数(即：散列函数)是散列表的关键部分，它将key与value建立映射关系，一般映射函数可以根据key的哈希值与Bucket数组大小取模得到，即 `key->h % ht->nTableSize` ，但是PHP却不是这么做的：

```
nIndex = key->h | ht->nTableMask;
```

显然位运算要比取模更快。

`nTableMask` 为 `nTableSize` 的负数，即: `nTableMask = -nTableSize` ，因为 `nTableSize` 等于2^n，所以 `nTableMask` 二进制位右侧全部为0，也就保证了nIndex落在数组索引的范围之内(`|nIndex| <= nTableSize`)：

```
11111111 11111111 11111111 11111000   -8
11111111 11111111 11111111 11110000   -16
11111111 11111111 11111111 11100000   -32
11111111 11111111 11111111 11000000   -64
11111111 11111111 11111111 10000000   -128
```

2.2.3 哈希碰撞

哈希碰撞是指不同的key可能计算得到相同的哈希值(数值索引的哈希值直接就是数值本身)，但是这些值又需要插入同一个散列表。一般解决方法是将Bucket串成链表，查找时遍历链表比较key。

PHP的实现也是如此，只是将链表的指针指向转化为了数值指向，即：指向冲突元素的指针并没有直接存在Bucket中，而是保存到了value的 `zval` 中：

```
struct _zval_struct {
    zend_value        value;            /* value */
    ...
    union {
        uint32_t     var_flags;
        uint32_t     next;                 /* hash collision cha
in */
        uint32_t     cache_slot;           /* literal cache slot
 */
        uint32_t     lineno;               /* line number (for a
st nodes) */
        uint32_t     num_args;             /* arguments number f
or EX(This) */
        uint32_t     fe_pos;               /* foreach position */

        uint32_t     fe_iter_idx;          /* foreach iterator i
ndex */
    } u2;
};
```

当出现冲突时将原value的位置保存到新value的 `zval.u2.next` 中，然后将新插入的value的位置更新到散列表，也就是后面冲突的value始终插入header。所以查找过程类似：

```
zend_ulong h = zend_string_hash_val(key);
uint32_t idx = ht->arHash[h & ht->nTableMask];
while (idx != INVALID_IDX) {
    Bucket *b = &ht->arData[idx];
    if (b->h == h && zend_string_equals(b->key, key)) {
        return b;
    }
    idx = Z_NEXT(b->val); //移到下一个冲突的value
}
return NULL;
```

2.2.4 插入、查找、删除

这几个基本操作比较简单，不再赘述，定位到元素所在Bucket位置后的操作类似单链表的插入、删除、查找。

2.2.5 扩容

散列表可存储的value数是固定的，当空间不够用时就要进行扩容了。

PHP散列表的大小为2^n，插入时如果容量不够则首先检查已删除元素所占比例，如果达到阈值(ht->nNumUsed - ht->nNumOfElements > (ht->nNumOfElements >> 5)，则将已删除元素移除，重建索引，如果未到阈值则进行扩容操作，扩大为当前大小的2倍，将当前Bucket数组复制到新的空间，然后重建索引。

```
//zend_hash.c
static void ZEND_FASTCALL zend_hash_do_resize(HashTable *ht)
{

    if (ht->nNumUsed > ht->nNumOfElements + (ht->nNumOfElements
>> 5)) {
        //只有到一定阈值才进行rehash操作
        zend_hash_rehash(ht); //重建索引数组
    } else if (ht->nTableSize < HT_MAX_SIZE) {
        //扩容
        void *new_data, *old_data = HT_GET_DATA_ADDR(ht);
        //扩大为2倍，加法要比乘法快，小的优化点无处不在...
        uint32_t nSize = ht->nTableSize + ht->nTableSize;
        Bucket *old_buckets = ht->arData;

        //新分配arData空间，大小为:(sizeof(Bucket) + sizeof(uint32_
t)) * nSize
        new_data = pemalloc(HT_SIZE_EX(nSize, -nSize), ...);
        ht->nTableSize = nSize;
        ht->nTableMask = -ht->nTableSize;
        //将arData指针偏移到Bucket数组起始位置
        HT_SET_DATA_ADDR(ht, new_data);
        //将旧的Bucket数组拷到新空间
        memcpy(ht->arData, old_buckets, sizeof(Bucket) * ht->nNu
mUsed);
        //释放旧空间
        pefree(old_data, ht->u.flags & HASH_FLAG_PERSISTENT);

        //重建索引数组：散列表
        zend_hash_rehash(ht);
        ...
    }
    ...
}

#define HT_SET_DATA_ADDR(ht, ptr) do { \
        (ht)->arData = (Bucket*)(((char*)(ptr)) + HT_HASH_SIZE((
ht)->nTableMask)); \
    } while (0)
```

2.2.6 重建散列表

当删除元素达到一定数量或扩容后都需要重建散列表，因为value在Bucket位置移动了或哈希数组nTableSize变化了导致key与value的映射关系改变，重建过程实际就是遍历Bucket数组中的value，然后重新计算映射值更新到散列表，除了更新散列表之外，这里还有一个重要的处理：移除已删除的value，开始的时候我们说过，删除value时只是将value的type设置为IS_UNDEF，并没有实际从Bucket数组中删除，如果这些value一直存在那么将浪费很多空间，所以这里会把它们移除，操作的方式也比较简单：将后面未删除的value依次前移，具体过程如下：

```
//zend_hash.c
ZEND_API int ZEND_FASTCALL zend_hash_rehash(HashTable *ht)
{
    Bucket *p;
    uint32_t nIndex, i;
    ...
    i = 0;
    p = ht->arData;
    if (ht->nNumUsed == ht->nNumOfElements) { //没有已删除的直接遍
历Bucket数组重新插入索引数组即可
        do {
            nIndex = p->h | ht->nTableMask;
            Z_NEXT(p->val) = HT_HASH(ht, nIndex);
            HT_HASH(ht, nIndex) = HT_IDX_TO_HASH(i);
            p++;
        } while (++i < ht->nNumUsed);
    } else {
        do {
            if (UNEXPECTED(Z_TYPE(p->val) == IS_UNDEF)) {
                //有已删除元素则将后面的value依次前移，压实Bucket数组
                ......
                while (++i < ht->nNumUsed) {
                    p++;
                    if (EXPECTED(Z_TYPE_INFO(p->val) != IS_UNDEF
)) {
                        ZVAL_COPY_VALUE(&q->val, &p->val);
                        q->h = p->h;
                        nIndex = q->h | ht->nTableMask;
                        q->key = p->key;
```

```
                        Z_NEXT(q->val) = HT_HASH(ht, nIndex);
                        HT_HASH(ht, nIndex) = HT_IDX_TO_HASH(j);
                        if (UNEXPECTED(ht->nInternalPointer == i
)) {
                            ht->nInternalPointer = j;
                        }
                        q++;
                        j++;
                    }
                }
                ......
                ht->nNumUsed = j;
                break;
            }

            nIndex = p->h | ht->nTableMask;
            Z_NEXT(p->val) = HT_HASH(ht, nIndex);
            HT_HASH(ht, nIndex) = HT_IDX_TO_HASH(i);
            p++;
        }while(++i < ht->nNumUsed);
    }
}
```

除了上面这些操作，PHP中关于HashTable的还有很多，这里不再介绍。

2.3 静态变量

PHP中局部变量分配在zend_execute_data结构上，每次执行zend_op_array都会生成一个新的zend_execute_data，局部变量在执行之初分配，然后在执行结束时释放，这是局部变量的生命周期，而局部变量中有一种特殊的类型：静态变量，它们不会在函数执行完后释放，当程序执行离开函数域时静态变量的值被保留下来，下次执行时仍然可以使用之前的值。

PHP中的静态变量通过 `static` 关键词创建：

```
function my_func(){
    static $count = 4;
    $count++;
    echo $count,"\n";
}
my_func();
my_func();
===========================
5
6
```

2.3.1 静态变量的存储

静态变量既然不会随执行的结束而释放，那么很容易想到它的保存位置： `zend_op_array->static_variables` ，这是一个哈希表，所以PHP中的静态变量与普通局部变量不同，它们没有分配在执行空间zend_execute_data上，而是以哈希表的形式保存在zend_op_array中。

> 静态变量只会初始化一次，注意：它的初始化发生在编译阶段而不是执行阶段，上面这个例子中： `static $count = 4;` 是在编译阶段发现定义了一个静态变量，然后插进了zend_op_array->static_variables中，并不是执行的时候把static_variables中的值修改为4，所以上面执行的时候会输出5、6，再次执行并没有重置静态变量的值。
>
> 这个特性也意味着静态变量初始的值不能是变量，比如： `static $count = $xxx;` 这样定义将会报错。

2.3.2 静态变量的访问

局部变量通过编译时确定的编号进行读写操作，而静态变量通过哈希表保存，这就使得其不能像普通变量那样有一个固定的编号，有一种可能是通过变量名索引的，那么究竟是否如此呢？我们分析下其编译过程。

静态变量编译的语法规则：

```
statement:
    ...
    |   T_STATIC static_var_list ';'    { $$ = $2; }
    ...
;

static_var_list:
        static_var_list ',' static_var { $$ = zend_ast_list_add($1, $3); }
    |   static_var { $$ = zend_ast_create_list(1, ZEND_AST_STMT_LIST, $1); }
;

static_var:
        T_VARIABLE          { $$ = zend_ast_create(ZEND_AST_STATIC, $1, NULL); }
    |   T_VARIABLE '=' expr { $$ = zend_ast_create(ZEND_AST_STATIC, $1, $3); }
;
```

语法解析后生成了一个 `ZEND_AST_STATIC` 语法树节点，接着再看下这个节点编译为opcode的过程：zend_compile_static_var。

```
void zend_compile_static_var(zend_ast *ast)
{
    zend_ast *var_ast = ast->child[0];
    zend_ast *value_ast = ast->child[1];
    zval value_zv;

    if (value_ast) {
        //定义了初始值
        zend_const_expr_to_zval(&value_zv, value_ast);
    } else {
        //无初始值
        ZVAL_NULL(&value_zv);
    }

    zend_compile_static_var_common(var_ast, &value_zv, 1);
}
```

这里首先对初始化值进行编译，最终得到一个固定值，然后调用：`zend_compile_static_var_common()` 处理，首先判断当前编译的 `zend_op_array->static_variables` 是否已创建，未创建则分配一个HashTable，接着将定义的静态变量插入：

```
//zend_compile_static_var_common():
if (!CG(active_op_array)->static_variables) {
    ALLOC_HASHTABLE(CG(active_op_array)->static_variables);
    zend_hash_init(CG(active_op_array)->static_variables, 8, NULL
, ZVAL_PTR_DTOR, 0);
}
//插入静态变量
zend_hash_update(CG(active_op_array)->static_variables, Z_STR(va
r_node.u.constant), value);
```

插入静态变量哈希表后并没有完成，接下来还有一个重要操作：

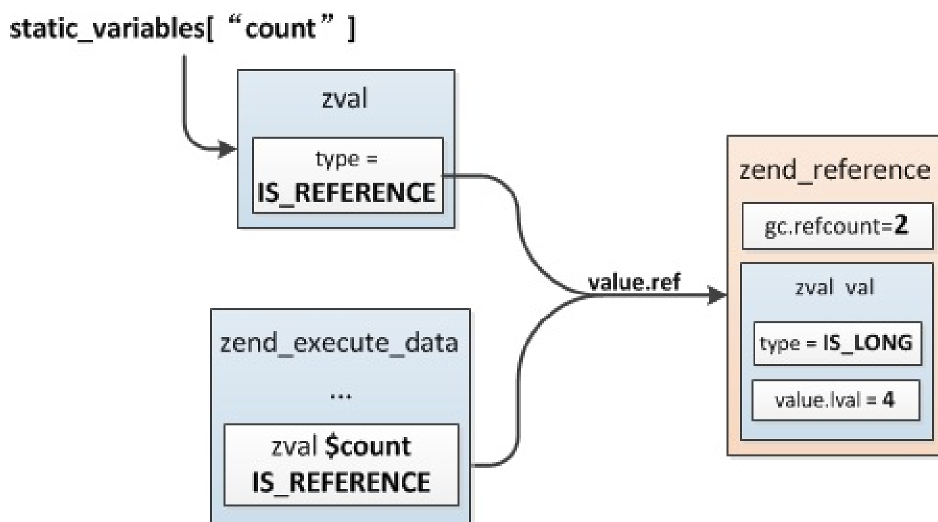
```
//生成一条ZEND_FETCH_W的opcode
opline = zend_emit_op(&result, by_ref ? ZEND_FETCH_W : ZEND_FETC
H_R, &var_node, NULL);
opline->extended_value = ZEND_FETCH_STATIC;

if (by_ref) {
    zend_ast *fetch_ast = zend_ast_create(ZEND_AST_VAR, var_ast)
;
    //生成一条ZEND_ASSIGN_REF的opcode
    zend_emit_assign_ref_znode(fetch_ast, &result);
}
```

后面生成了两条opcode：

- **ZEND_FETCH_W:** 这条opcode对应的操作是创建一个IS_INDIRECT类型的zval，指向static_variables中对应静态变量的zval
- **ZEND_ASSIGN_REF:** 它的操作是引用赋值，即将一个引用赋值给CV变量

通过上面两条opcode可以确定静态变量的读写过程：首先根据变量名在static_variables中取出对应的zval，然后将它修改为引用类型并赋值给局部变量，也就是说 `static $count = 4;` 包含了两个操作，严格的将 `$count` 并不是真正的静态变量，它只是一个指向静态变量的局部变量，执行时实际操作是： `$count = & static_variables["count"];` 。上面例子$count与static_variables["count"]间的关系如图所示。



2.4 全局变量

PHP中把定义在函数、类之外的变量称之为全局变量，也就是定义在主脚本中的变量，这些变量可以在函数、成员方法中通过global关键字引入使用。

```
function test() {
    global $id;
    $id++;
}

$id = 1;
test();
echo $id;
```

2.4.1 全局变量初始化

全局变量在整个请求执行期间始终存在，它们保存在 `EG(symbol_table)` 中，也就是全局变量符号表，与静态变量的存储一样，这也是一个哈希表，主脚本(或include、require)在 `zend_execute_ex` 执行开始之前会把当前作用域下的所有局部变量添加到 `EG(symbol_table)` 中，这一步操作后面介绍zend执行过程时还会讲到，这里先简单提下：

```
ZEND_API void zend_execute(zend_op_array *op_array, zval *return
_value)
{
    ...
    i_init_execute_data(execute_data, op_array, return_value);
    zend_execute_ex(execute_data);
    ...
}
```

`i_init_execute_data()` 这个函数中会把局部变量插入到EG(symbol_table)：

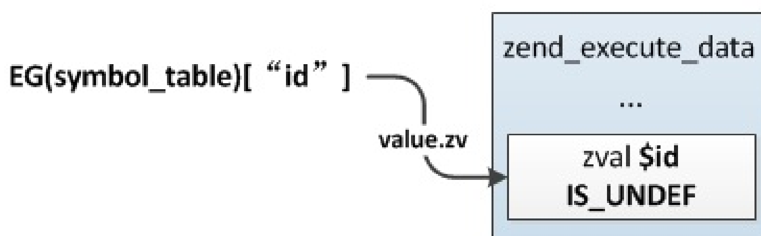
```
ZEND_API void zend_attach_symbol_table(zend_execute_data *execut
e_data)
{
    zend_op_array *op_array = &execute_data->func->op_array;
    HashTable *ht = execute_data->symbol_table;

    if (!EXPECTED(op_array->last_var)) {
        return;
    }

    zend_string **str = op_array->vars;
    zend_string **end = str + op_array->last_var;
    //局部变量数组起始位置
    zval *var = EX_VAR_NUM(0);

    do{
        zval *zv = zend_hash_find(ht, *str);
        //插入全局变量符号表
        zv = zend_hash_add_new(ht, *str, var);
        //哈希表中value指向局部变量的zval
        ZVAL_INDIRECT(zv, var);
        ...
    }while(str != end);
}
```

从上面的过程可以很直观的看到，在执行前遍历局部变量，然后插入EG(symbol_table)，EG(symbol_table)中的value直接指向局部变量的zval，示例经过这一步的处理之后(此时局部变量只是分配了zval，但还未初始化，所以是IS_UNDEF)：

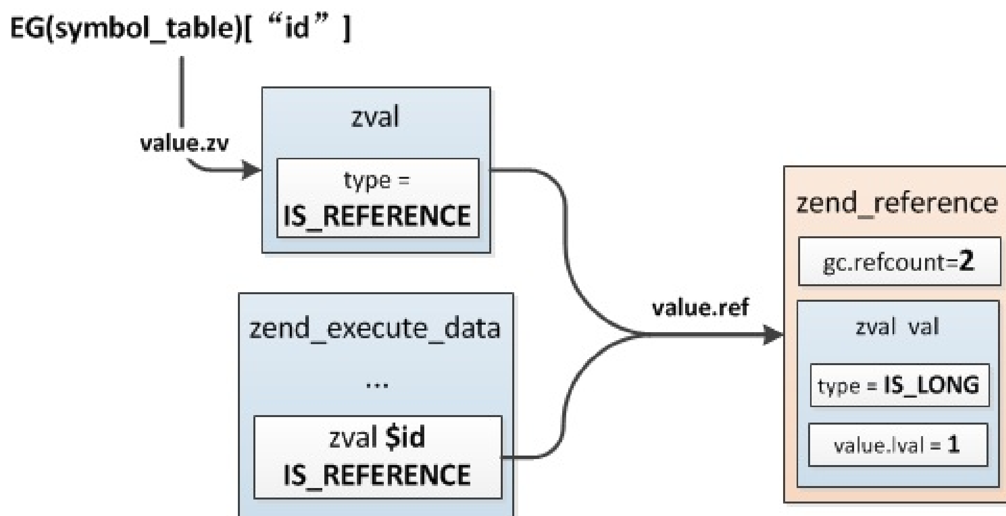


2.4.2 全局变量的访问

与静态变量的访问一样，全局变量也是将原来的值转换为引用，然后在global导入的作用域内创建一个局部变量指向该引用：

```
global $id; // 相当于：$id = & EG(symbol_table)["id"];
```

具体的操作过程不再细讲，与静态变量的处理过程一致，这时示例中局部变量与全局变量的引用情况如下图。



2.4.3 超全局变量

全局变量除了通过global引入外还有一类特殊的类型，它们不需要使用global引入而可以直接使用，这些全局变量称为：超全局变量。

超全局变量实际是PHP内核定义的一些全局变量：$GLOBALS、$_SERVER、$_REQUEST、$_POST、$_GET、$_FILES、$_ENV、$_COOKIE、$_SESSION、argv、argc。

2.4.4 销毁

局部变量如果没有手动销毁，那么在函数执行结束时会将它们销毁，而全局变量则是在整个请求结束时才会销毁，即使是我们直接在PHP脚本中定义在函数外的那些变量。

```
void shutdown_destructors(void)
{
    if (CG(unclean_shutdown)) {
        EG(symbol_table).pDestructor = zend_unclean_zval_ptr_dtor;
    }
    zend_try {
        uint32_t symbols;
        do {
            symbols = zend_hash_num_elements(&EG(symbol_table));
            //销毁
            zend_hash_reverse_apply(&EG(symbol_table), (apply_func_t) zval_call_destructor);
        } while (symbols != zend_hash_num_elements(&EG(symbol_table)));
    }
    ...
}
```

2.5 常量

常量是一个简单值的标识符（名字）。如同其名称所暗示的，在脚本执行期间该值不能改变。常量默认为大小写敏感。通常常量标识符总是大写的。

常量名和其它任何 PHP 标签遵循同样的命名规则。合法的常量名以字母或下划线开始，后面跟着任何字母，数字或下划线。

PHP中的常量通过 `define()` 函数定义：

```
define('CONST_VAR_1', 1234);
```

2.5.1 常量的存储

在内核中常量存储在 `EG(zend_constants)` 哈希表中，访问时也是根据常量名直接到哈希表中查找，其实现比较简单。

常量的数据结构：

```
typedef struct _zend_constant {
    zval value;    //常量值
    zend_string *name; //常量名
    int flags;   //常量标识位
    int module_number; //所属扩展、模块
} zend_constant;
```

常量的几个属性都比较直观，这里只介绍下flags，它的值可以是以下三个中任意组合：

```
#define CONST_CS                (1<<0)   //大小写敏感
#define CONST_PERSISTENT        (1<<1)   //持久化的
#define CONST_CT_SUBST          (1<<2)   //允许编译时替换
```

介绍下三种flag代表的含义：

- **CONST_CS:** 大小写敏感，默认是开启的，用户通过define()定义的始终是区

分大小写的，通过扩展定义的可以自由选择

- **CONST_PERSISTENT:** 持久化的，只有通过扩展、内核定义的才支持，这种常量不会在request结束时清理掉
- **CONST_CT_SUBST:** 允许编译时替换，编译时如果发现有地方在读取常量的值，那么编译器会尝试直接替换为常量值，而不是在执行时再去读取，目前这个flag只有TRUE、FALSE、NULL三个常量在使用

2.5.2 常量的销毁

非持久化常量在request请求结束时销毁，具体销毁操作在：`php_request_shutdown()->zend_deactivate()->shutdown_executor()->clean_non_persistent_constants()`。

```
void clean_non_persistent_constants(void)
{
    if (EG(full_tables_cleanup)) {
        zend_hash_apply(EG(zend_constants), clean_non_persistent
_constant_full);
    } else {
        zend_hash_reverse_apply(EG(zend_constants), clean_non_pe
rsistent_constant);
    }
}
```

然后从哈希表末尾开始向前遍历EG(zend_constants)，将非持久化常量删除，直到碰到第一个持久化常量时，停止遍历，正常情况下所有通过扩展定义的常量一定是在PHP中通过define定义之前，当然也并非绝对，这里只是说在所有常量均是在MINT阶段定义的情况。

持久化常量是在 `php_module_shutdown()` 阶段销毁的，具体过程与上面类似。

3.1 PHP代码的编译

PHP是解析型高级语言，事实上从Zend内核的角度来看PHP就是一个普通的C程序，它有main函数，我们写的PHP代码是这个程序的输入，然后经过内核的处理输出结果，内核将PHP代码"翻译"为C程序可识别的过程就是PHP的编译。

那么这个"翻译"过程具体都有哪些操作呢？

C程序在编译时将一行行代码编译为机器码，每一个操作都认为是一条机器指令，这些指令写入到编译后的二进制程序中，执行的时候将二进制程序load进相应的内存区域(常量区、数据区、代码区)、分配运行栈，然后从代码区起始位置开始执行，这是C程序编译、执行的简单过程。

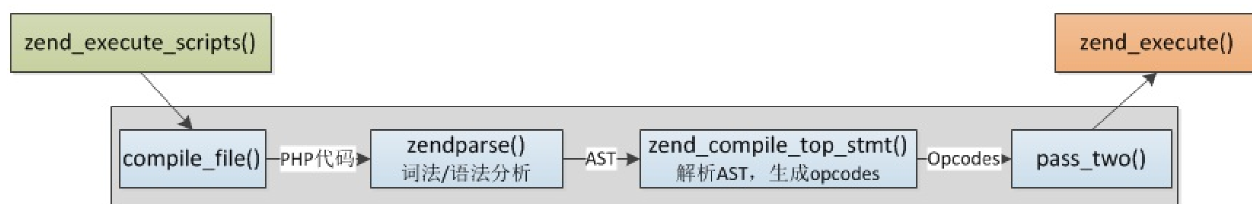
同样，PHP的编译与普通的C程序类似，只是PHP代码没有编译成机器码，而是解析成了若干条opcode数组，每条opcode就是C里面普通的struct，含义对应C程序的机器指令，执行的过程就是引擎依次执行opcode，比如我们在PHP里定义一个变量: `$a = 123;` ，最终到内核里执行就是malloc一块内存，然后把值写进去。

所以PHP的解析过程任务就是将PHP代码转化为opcode数组，代码里的所有信息都保存在opcode中，然后将opcode数组交给zend引擎执行，opcode就是内核具体执行的命令，比如赋值、加减操作、函数调用等，每一条opcode都对应一个处理handle，这些handler是提前定义好的C函数。

从PHP代码到opcode是怎么实现的？最容易想到的方式就是正则匹配，当然过程没有这么简单。PHP编译过程包括词法分析、语法分析，使用re2c、bison完成，旧的PHP版本直接生成了opcode，PHP7新增了抽象语法树（AST），在语法分析阶段生成AST，然后再生成opcode数组。



PHP编译阶段的基本过程如下图：



后面两个小节将看下 **PHP**代码**->AST->Opcodes** 的具体编译过程。

3.1.1 词法解析、语法解析

这一节我们分析下PHP的解析阶段，即 **PHP代码->抽象语法树(AST)** 的过程。

PHP使用re2c、bison完成这个阶段的工作:

- **re2c:** 词法分析器，将输入分割为一个个有意义的词块，称为token
- **bison:** 语法分析器，确定词法分析器分割出的token是如何彼此关联的

例如：

```
$a = 2 + 3;
```

词法分析器将上面的语句分解为这些token：$a、=、2、+、3，接着语法分析器确定了 `2+3` 是一个表达式，而这个表达式被赋值给了 `a` ，我们可以这样定义词法解析规则：

```
/*!re2c
    LABEL   [a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*
    LNUM    [0-9]+

    //规则
    "$"{LABEL} {return T_VAR;}
    {LNUM} {return T_NUM;}
*/
```

然后定义语法解析规则：

```
//token定义
%token T_VAR
%token T_NUM

//语法规则
statement:
    T_VAR '=' T_NUM '+' T_NUM {ret = str2int($3) + str2int($5);p
rintf("%d",ret);}
;
```

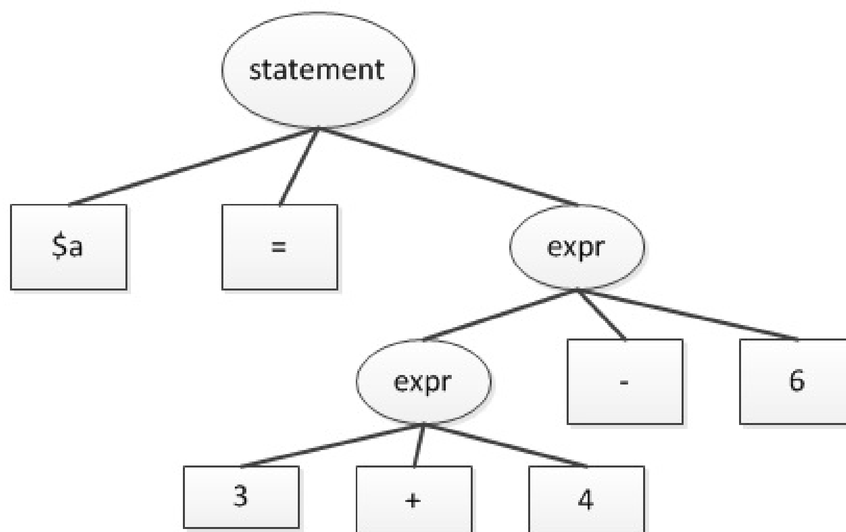
上面的语法规则只能识别两个数值相加，假如我们希望支持更复杂的运算，比如：

```
$a = 3 + 4 - 6;
```

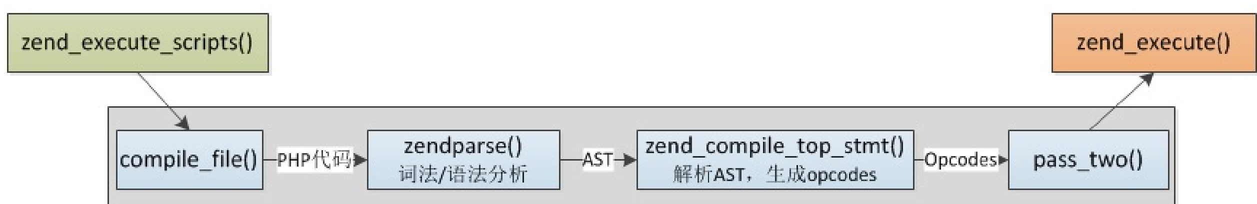
则可以配置递归规则：

```
//语法规则
statement:
    T_VAR '=' expr {}
;
expr:
    T_NUM {...}
    |expr '?' T_NUM {}
;
```

这样将支持若干表达式，用语法分析树表示：



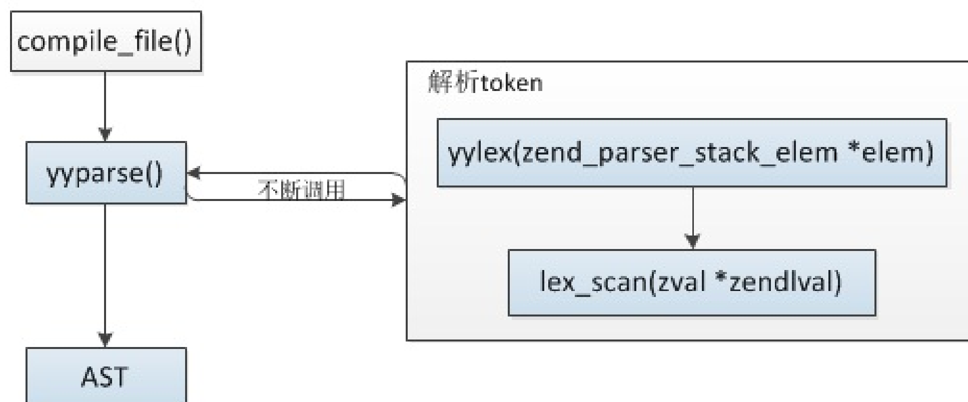
接下来我们看下PHP具体的解析过程，PHP编译阶段流程：



其中 **zendparse()** 就是词法、语法解析过程，这个函数实际就是bison中提供的语法解析函数 **yyparse()** ：

```
#define yyparse         zendparse
```

yyparse() 不断调用 **yylex()** 得到token，然后根据token匹配语法规则：



```
#define yylex           zendlex

//zend_compile.c
int zendlex(zend_parser_stack_elem *elem)
{
    zval zv;
    int retval;
    ...

again:
    ZVAL_UNDEF(&zv);
    retval = lex_scan(&zv);
    if (EG(exception)) {
        //语法错误
        return T_ERROR;
    }
    ...

    if (Z_TYPE(zv) != IS_UNDEF) {
        //如果在分割token中有zval生成则将其值复制到zend_ast_zval结构中
        elem->ast = zend_ast_create_zval(&zv);
    }

    return retval;
}
```

这里两个关键点需要注意：

(1) token值：词法解析器解析到的token值内容就是token值，这些值统一通过 **zval** 存储，上面的过程中可以看到调用lex_scan参数是是个zval*，在具体的命中规则总会将解析到的token保存到这个值，从而传递给语法解析器使用，比如PHP中的解析变量的规则：`$a;`，其词法解析规则为：

```
<ST_IN_SCRIPTING,ST_DOUBLE_QUOTES,ST_HEREDOC,ST_BACKQUOTE,ST_VAR
_OFFSET>"$"{LABEL} {
    //将匹配到的token值保存在zval中
    zend_copy_value(zendlval, (yytext+1), (yyleng-1)); //只保存{L
ABEL}内容，不包括$，所以是yytext+1
    RETURN_TOKEN(T_VARIABLE);
}
```

zendlval就是我们传入的zval*，yytext指向命中的token值起始位置，yyleng为token值的长度。

(2) 语义值类型：bison调用re2c分割token有两个含义，第一个是token类型，另一个是token值，token类型一般以yylex的返回值告诉bison，而token值就是语义值，这个值一般定义为固定的类型，这个类型就是语义值类型，默认为int，可以通过 **YYSTYPE** 定义，而PHP中这个类型是 **zend_parser_stack_elem** ，这就是为什么zendlex的参数为 `zend_parser_stack_elem` 的原因。

```
#define YYSTYPE zend_parser_stack_elem

typedef union _zend_parser_stack_elem {
    zend_ast *ast; //抽象语法树主要结构
    zend_string *str;
    zend_ulong num;
} zend_parser_stack_elem;
```

实际这是个union，ast类型用的比较多(其它两种类型暂时没发现有地方在用)，这样可以通过%token、%type将对应的值修改为elem.ast，所以在zend*languageparser.y*中使用的*$$*、*$1*、*$2......*多数都是*__zend_parser_stack_elem.ast* ：

```
%token <ast> T_LNUMBER   "integer number (T_LNUMBER)"
%token <ast> T_DNUMBER   "floating-point number (T_DNUMBER)"
%token <ast> T_STRING    "identifier (T_STRING)"
%token <ast> T_VARIABLE  "variable (T_VARIABLE)"

%type <ast> top_statement namespace_name name statement function
_declaration_statement
%type <ast> class_declaration_statement trait_declaration_statem
ent
%type <ast> interface_declaration_statement interface_extends_li
st
```

语法解析器从start开始调用，然后层层匹配各个规则，语法解析器根据命中的语法规则创建AST节点，最后将生成的AST根节点赋到 **CG(ast)** ：

```
%% /* Rules */

start:
    top_statement_list  { CG(ast) = $1; }
;

top_statement_list:
    top_statement_list top_statement { $$ = zend_ast_list_add($1
, $2); }
    |   /* empty */ { $$ = zend_ast_create_list(0, ZEND_AST_STMT
_LIST); }
;
```

首先会创建一个根节点list，然后将后面不断命中top_statement生成的ast加到这个list中，zend_ast具体结构：

```
enum _zend_ast_kind {
    ZEND_AST_ZVAL = 1 << ZEND_AST_SPECIAL_SHIFT,
    ZEND_AST_ZNODE,

    /* list nodes */
    ZEND_AST_ARG_LIST = 1 << ZEND_AST_IS_LIST_SHIFT,
    ...
};

struct _zend_ast {
    zend_ast_kind kind; /* Type of the node (ZEND_AST_* enum constant) */
    zend_ast_attr attr; /* Additional attribute, use depending on node type */
    uint32_t lineno;    /* Line number */
    zend_ast *child[1]; /* Array of children (using struct hack) */
};

typedef struct _zend_ast_list {
    zend_ast_kind kind;
    zend_ast_attr attr;
    uint32_t lineno;
    uint32_t children;
    zend_ast *child[1];
} zend_ast_list;
```

根节点实际为zend_ast_list，每条语句对应的ast保存在child中，使用中zend_ast_list、zend_ast可以相互转化，kind标识的是ast节点类型，后面会根据这个值生成具体的opcode，另外函数、类还会用到另外一种ast节点结构：

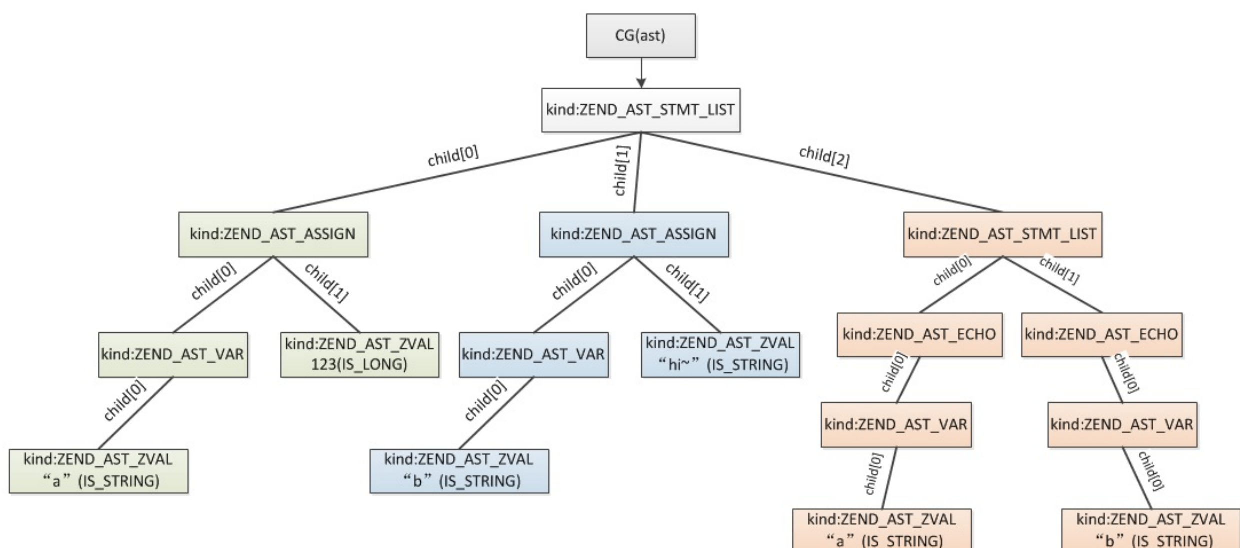
```
typedef struct _zend_ast_decl {
    zend_ast_kind kind;
    zend_ast_attr attr; /* Unused - for structure compatibility
*/
    uint32_t start_lineno; //开始行号
    uint32_t end_lineno;   //结束行号
    uint32_t flags;
    unsigned char *lex_pos;
    zend_string *doc_comment;
    zend_string *name;
    zend_ast *child[4]; //类中会将继承的父类、实现的接口以及类中的语句解
析保存在child中
} zend_ast_decl;
```

这么看比较难理解，接下来我们从一个简单的例子看下最终生成的语法树。

```
$a = 123;
$b = "hi~";

echo $a,$b;
```

具体解析过程这里不再解释，有兴趣的可以翻下zend_language_parse.y中，这个过程不太容易理解，需要多领悟几遍，最后生成的ast如下图：



总结：

这一节我们主要介绍了PHP词法、语法解析生成抽象语法树(AST)的过程，此过程是PHP语法实现的基础，也是zend引擎非常关键的一部分，后续介绍的内容都是基于此过程的产出结果展开的。这部分内容关键在于对re2c、bison的应用上，如果是初次接触它们可能不太容易理解，这里不再对re2c、bison作更多解释，想要了解更多的推荐看下 《**flex与bison**》 这本书。

3.1.2 抽象语法树编译流程

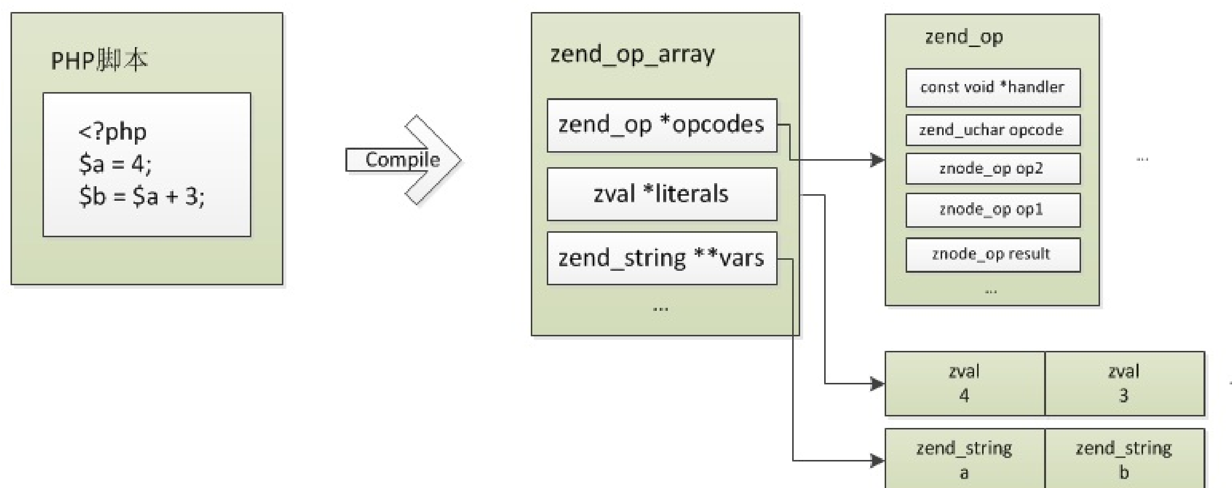
上一小节我们简单介绍了从PHP代码解析为抽象语法树的过程，这一节我们再介绍下从 抽象语法树**->Opcodes** 的过程。

语法解析过程的产物保存于CG(AST)，接着zend引擎会把AST进一步编译为 **zend_op_array** ，它是编译阶段最终的产物，也是执行阶段的输入，后面我们介绍的东西基本都是围绕zend*oparray*展开的，*AST*解析过程确定了当前脚本定义了哪些变量，并为这些变量 __顺序编号 ，这些值在使用时都是按照这个编号获取的，另外也将变量的初始化值、调用的函数/类/常量名称等值(称之为字面量)保存到zend_op_array.literals中，这些字面量也有一个唯一的编号，所以执行的过程实际就是根据各指令调用不同的C函数，然后根据变量、字面量、临时变量的编号对这些值进行处理加工。

我们首先看下zend_op_array的结构，明确几个关键信息，然后再看下ast编译为zend_op_array的过程。

3.1.2.1 zend_op_array数据结构

PHP主脚本会生成一个zend_op_array，每个function也会编译为独立的zend_op_array，所以从二进制程序的角度看zend_op_array包含着当前作用域下的所有堆栈信息，函数调用实际就是不同zend_op_array间的切换。




```
struct _zend_op_array {
    //common是普通函数或类成员方法对应的opcodes快速访问时使用的字段，后面分析PHP函数实现的时候会详细讲
```

```
    ...

    uint32_t *refcount;

    uint32_t this_var;

    uint32_t last;
    //opcode指令数组
    zend_op *opcodes;

    //PHP代码里定义的变量数：op_type为IS_CV的变量，不含IS_TMP_VAR、IS_VAR的
    //编译前此值为0，然后发现一个新变量这个值就加1
    int last_var;
    //临时变量数:op_type为IS_TMP_VAR、IS_VAR的变量
    uint32_t T;
    //PHP变量名数组
    zend_string **vars; //这个数组在ast编译期间配合last_var用来确定各个变量的编号，非常重要的一步操作
    ...

    //静态变量符号表:通过static声明的
    HashTable *static_variables;
    ...

    //字面量数量
    int last_literal;
    //字面量(常量)数组，这些都是在PHP代码定义的一些值
    zval *literals;

    //运行时缓存数组大小
    int  cache_size;
    //运行时缓存，主要用于缓存一些znode_op以便于快速获取数据，后面单独介绍这个机制
    void **run_time_cache;

    void *reserved[ZEND_MAX_RESERVED_RESOURCES];
};
```

zend_op_array.opcodes指向指令列表，具体每条指令的结构如下：

```
struct _zend_op {
    const void *handler; //指令执行handler
    znode_op op1;    //操作数1
    znode_op op2;    //操作数2
    znode_op result; //返回值
    uint32_t extended_value;
    uint32_t lineno;
    zend_uchar opcode;  //opcode指令
    zend_uchar op1_type; //操作数1类型
    zend_uchar op2_type; //操作数2类型
    zend_uchar result_type; //返回值类型
};

//操作数结构
typedef union _znode_op {
    uint32_t      constant;
    uint32_t      var;
    uint32_t      num;
    uint32_t      opline_num; /*  Needs to be signed */
    uint32_t      jmp_offset;
} znode_op;
```

opcode各字段含义下面展开说明。

3.1.2.1.1 handler

handler为每条opcode对应的C语言编写的 处理过程 ，所有opcode对应的处理过程定义在 `zend_vm_def.h` 中，值得注意的是这个文件并不是编译时用到的，因为opcode的 处理过程 有三种不同的提供形式：CALL、SWITCH、GOTO，默认方式为CALL，这个是什么意思呢？

每个opcode都代表了一些特定的处理操作，这个东西怎么提供呢？一种是把每种opcode负责的工作封装成一个function，然后执行器循环执行即可，这就是CALL模式的工作方式；另外一种是把所有opcode的处理方式通过C语言里面的label标签区分开，然后执行器执行的时候goto到相应的位置处理，这就是GOTO模式的工作方式；最后还有一种方式是把所有的处理方式写到一个switch下，然后通过case不同的opcode执行具体的操作，这就是SWITCH模式的工作方式。

假设opcode数组是这个样子：

```
int op_array[] = {
    opcode_1,
    opcode_2,
    opcode_3,
    ...
};
```

各模式下的工作过程类似这样：

```
//CALL模式
void opcode_1_handler() {...}

void opcode_2_handler() {...}
...

void execute(int []op_array)
{
    void *opcode_handler_list[] = {&opcode_1_handler, &opcode_2_
handler, ...};

    while(1){
        void handler = opcode_handler_list[op_array[i]];
        handler(); //call handler
        i++;
    }
}

//GOTO模式
void execute(int []op_array)
{
    while(1){
        goto opcode_xx_handler_label;
    }

opcode_1_handler_label:
    ...

opcode_2_handler_label:
    ...
```

```
    ...
}

//SWITCH模式
void execute(int []op_array)
{
    while(1){
        switch(op_array[i]){
            case opcode_1:
                ...
            case opcode_2:
                ...
            ...
        }

        i++;
    }
}
```

三种模式效率是不同的，GOTO最快，怎么选择其它模式呢？下载PHP源码后不要直接编译，Zend目录下有个文件： `zend_vm_gen.php` ，在编译PHP前执行： `php zend_vm_gen.php --with-vm-kind=CALL|SWITCH|GOTO` ，这个脚本将重新生成: `zend_vm_opcodes.h` 、 `zend_vm_opcodes.c` 、 `zend_vm_execute.h` 三个文件覆盖原来的，然后再编译PHP即可。

后面分析的过程使用的都是默认模式 `CALL` ，也就是opcode对应的handler为一个函数指针，编译时opcode对应的handler是如何根据opcode索引到的呢？

opcode的数值各不相同，同时可以根据两个zend_op的类型设置不同的处理handler，因此每个opcode指令最多有20个（25去掉重复的5个）对应的处理handler，所有的handler按照opcode数值的顺序定义在一个大数组中: `zend_opcode_handlers` ，每25个为同一个opcode，如果对应的op_type类型handler则可以设置为空：

```
//zend_vm_execute.h
void zend_init_opcodes_handlers(void)
{
    static const void *labels[] = {
        ZEND_NOP_SPEC_HANDLER,
        ZEND_NOP_SPEC_HANDLER,
        ...
    };
    zend_opcode_handlers = labels;
}
```

索引的算法：

```
//zend_vm_execute.h
static const void *zend_vm_get_opcode_handler(zend_uchar opcode,
const zend_op* op)
{
        //因为op_type为2的倍数，所以这里做了下转化，转成了0-4
        static const int zend_vm_decode[] = {
            _UNUSED_CODE, /* 0              */
            _CONST_CODE,  /* 1 = IS_CONST   */
            _TMP_CODE,    /* 2 = IS_TMP_VAR */
            _UNUSED_CODE, /* 3              */
            _VAR_CODE,    /* 4 = IS_VAR     */
            _UNUSED_CODE, /* 5              */
            _UNUSED_CODE, /* 6              */
            _UNUSED_CODE, /* 7              */
            _UNUSED_CODE, /* 8 = IS_UNUSED  */
            _UNUSED_CODE, /* 9              */
            _UNUSED_CODE, /* 10             */
            _UNUSED_CODE, /* 11             */
            _UNUSED_CODE, /* 12             */
            _UNUSED_CODE, /* 13             */
            _UNUSED_CODE, /* 14             */
            _UNUSED_CODE, /* 15             */
            _CV_CODE      /* 16 = IS_CV     */
        };
        //根据op1_type、op2_type、opcode得到对应的handler
        return zend_opcode_handlers[opcode * 25 + zend_vm_decode
```

```
[op->op1_type] * 5 + zend_vm_decode[op->op2_type]];
}

ZEND_API void zend_vm_set_opcode_handler(zend_op* op)
{
    //设置zend_op的handler，这个操作是在编译期间完成的
    op->handler = zend_vm_get_opcode_handler(zend_user_opcodes[o
p->opcode], op);
}

#define _CONST_CODE  0
#define _TMP_CODE    1
#define _VAR_CODE    2
#define _UNUSED_CODE 3
#define _CV_CODE     4
```

3.1.2.1.2 操作数(znode_op)

操作数类型实际就是个32位整形，它主要用于存储一些变量的索引位置、数值记录等等。

```
typedef union _znode_op {
    uint32_t      constant;
    uint32_t      var;
    uint32_t      num;
    uint32_t      opline_num; /*  Needs to be signed */
    uint32_t      jmp_offset;
} znode_op;
```

每条opcode都有两个操作数(不一定都用到)，操作数记录着当前指令的关键信息，可以用于变量的存储、访问，比如赋值语句："$a = 45;",两个操作数分别记录"$a"、"45"的存储位置，执行时根据op2取到值"45"，然后赋值给"$a"，而"$a"的位置通过op1获取到。当然操作数并不是全部这么用的，上面只是赋值时候的情况，其它操作会有不同的用法，如函数调用时的传参，op1记录的就是传递的参数是第几个，op2记录的是参数的存储位置，result记录的是函数接收参数的存储位置。

3.1.2.1.3 操作数类型(op_type)

每个操作都有5种不同的类型：

```
#define IS_CONST    (1<<0)  //1
#define IS_TMP_VAR  (1<<1)  //2
#define IS_VAR      (1<<2)  //4
#define IS_UNUSED   (1<<3)  //8
#define IS_CV       (1<<4)  //16
```

- IS_CONST：字面量，编译时就可确定且不会改变的值，比如:$a = "hello~"，其中字符串"hello~"就是常量
- IS_TMP_VAR：临时变量，比如：$a = "hello~" . time()，其中 `"hello~" . time()` 的值类型就是IS_TMP_VAR，再比如:$a = "123" + $b， `"123" + $b` 的结果类型也是IS_TMP_VAR，从这两个例子可以猜测，临时变量多是执行期间其它类型组合现生成的一个中间值，由于它是现生成的，所以把IS_TMP_VAR赋值给IS_CV变量时不会增加其引用计数
- IS_VAR：PHP变量，这个很容易认为是PHP脚本里的变量，其实不是，这里PHP变量的含义可以这样理解：PHP变量是没有显式的在PHP脚本中定义的，不是直接在代码通过 `$var_name` 定义的。这个类型最常见的例子是PHP函数的返回值，再如 `$a[0]` 数组这种，它取出的值也是 `IS_VAR` ，再比如 `$$a` 这种
- IS_UNUSED：表示操作数没有用
- IS_CV：PHP脚本变量，即脚本里通过 `$var_name` 定义的变量，这些变量是编译阶段确定的，所以是compile variable，

`result_type` 除了上面几种类型外还有一种类型 `EXT_TYPE_UNUSED (1<<5)` ，返回值没有使用时会用到，这个跟 `IS_UNUSED` 的区别是： `IS_UNUSED` 表示本操作返回值没有意义(也可简单的认为没有返回值)，而 `EXT_TYPE_UNUSED` 的含义是有返回值，但是没有用到，比如函数返回值没有接收。

3.1.2.1.4 字面量、变量的存储

我们先想一下C程序是如何读写字面量、变量的。

```
#include <stdio.h>
int main()
{
    char *name = "pangudashu";

    printf("%s\n", name);
    return 0;
}
```

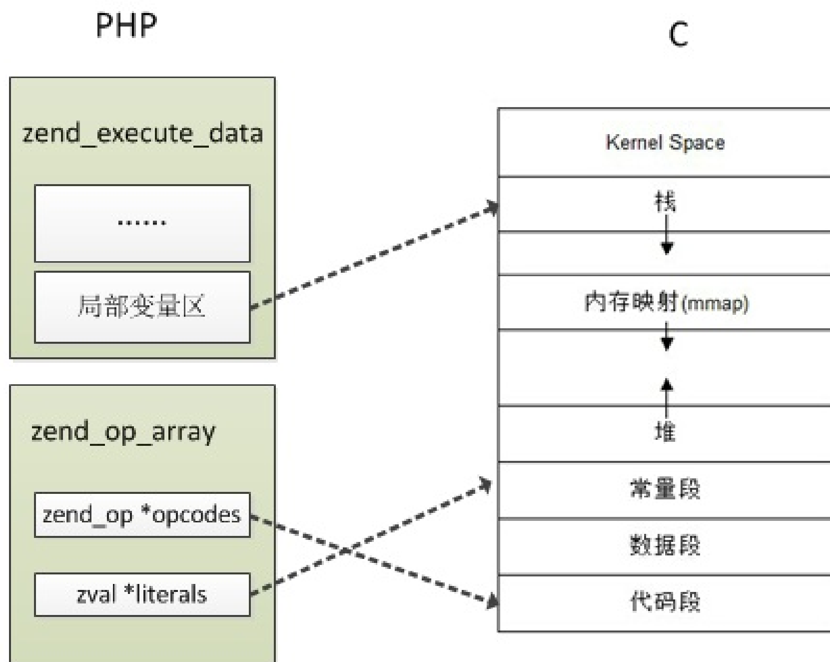
我们知道指针name分配在栈上，而"pangudashu"分配在常量区，那么"name"变量名分配在哪呢？

实际上C里面是不会存变量名称的，编译的过程会将变量名替换为偏移量表示： `ebp - 偏移量` 或 `esp + 偏移量` ，将上面的代码转为汇编:

```
.LC0:
    .string "pangudashu"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movq    $.LC0, -8(%rbp)
    movq    -8(%rbp), %rax
    movq    %rax, %rdi
    call    puts
    movl    $0, %eax
    leave
```

可以看到 `movq $.LC0, -8(%rbp)` ，而 `-8(%rbp)` 就是name变量。

虽然PHP代码不会直接编译为机器码，但编译、执行的设计跟C程序是一致的，也有常量区、变量也通过偏移量访问、也有虚拟的执行栈。



在编译时就可确定且不会改变的量称为字面量，也称作常量(IS_CONST)，这些值在编译阶段就已经分配zval，保存在 `zend_op_array->literals` 数组中(对应c程序的常量存储区)，访问时通过 `_zend_op_array->literals + 偏移量` 读取，举个例子：

```
<?php
$a = 56;
$b = "hello";
```

`56` 通过 `(zval*)(_zend_op_array->literals + 0)` 取到， `hello` 通过 `(zval*)(_zend_op_array->literals + 16)` 取到,具体变量的读写操作将在执行阶段详细分析，这里只分析编译阶段的操作。

3.1.2.2 AST->zend_op_array

上面我们介绍了zend_op_array结构，接下来我们回过头去看下语法解析(zendparse())之后的流程:

```
ZEND_API zend_op_array *compile_file(zend_file_handle *file_hand
le, int type)
{
    zend_op_array *op_array = NULL; //编译出的opcodes
    ...
```

```
    if (open_file_for_scanning(file_handle)==FAILURE) {//文件打开失败
        ...
    } else {
        zend_bool original_in_compilation = CG(in_compilation);
        CG(in_compilation) = 1;

        CG(ast) = NULL;
        CG(ast_arena) = zend_arena_create(1024 * 32);
        if (!zendparse()) { //语法解析
            zval retval_zv;
            zend_file_context original_file_context; //保存原来的zend_file_context
            zend_oparray_context original_oparray_context; //保存原来的zend_oparray_context，编译期间用于记录当前zend_op_array的opcodes、vars等数组的总大小
            zend_op_array *original_active_op_array = CG(active_op_array);
            op_array = emalloc(sizeof(zend_op_array)); //分配zend_op_array结构
            init_op_array(op_array, ZEND_USER_FUNCTION, INITIAL_OP_ARRAY_SIZE);//初始化op_array
            CG(active_op_array) = op_array; //将当前正在编译op_array指向当前
            ZVAL_LONG(&retval_zv, 1);

            if (zend_ast_process) {
                zend_ast_process(CG(ast));
            }

            zend_file_context_begin(&original_file_context); //初始化CG(file_context)
            zend_oparray_context_begin(&original_oparray_context); //初始化CG(context)
            zend_compile_top_stmt(CG(ast)); //AST->zend_op_array编译流程
            zend_emit_final_return(&retval_zv); //设置最后的返回值
            op_array->line_start = 1;
            op_array->line_end = CG(zend_lineno);
```

```
            pass_two(op_array);
            zend_oparray_context_end(&original_oparray_context);
            zend_file_context_end(&original_file_context);

            CG(active_op_array) = original_active_op_array;
        }
        ...
    }
    ...

    return op_array;
}
```

compile_file()操作中有几个保存原来值的操作，这是因为这个函数在PHP脚本执行中并不会只执行一次，主脚本执行时会第一次调用，而include、require也会调用，所以需要先保存当前值，然后执行完再还原回去。

AST->zend*oparray*编译是在 *__zend_compile_top_stmt()* 中完成，这个函数是总入口，会被多次递归调用：

```
//zend_compile.c
void zend_compile_top_stmt(zend_ast *ast)
{
    if (!ast) {
        return;
    }

    if (ast->kind == ZEND_AST_STMT_LIST) { //第一次进来一定是这种类型

        zend_ast_list *list = zend_ast_get_list(ast);
        uint32_t i;
        for (i = 0; i < list->children; ++i) {
            zend_compile_top_stmt(list->child[i]);//list各child语
句相互独立，递归编译
        }
        return;
    }

    //各语句编译入口
    zend_compile_stmt(ast);

    if (ast->kind != ZEND_AST_NAMESPACE && ast->kind != ZEND_AST
_HALT_COMPILER) {
        zend_verify_namespace();
    }
    //function、class两种情况的处理，非常关键的一步操作，后面分析函数、类
实现的章节再详细分析
    if (ast->kind == ZEND_AST_FUNC_DECL || ast->kind == ZEND_AST
_CLASS) {
        CG(zend_lineno) = ((zend_ast_decl *) ast)->end_lineno;
        zend_do_early_binding(); //很重要!!!
    }
}
```

首先从AST的根节点开始编译，根节点类型为ZEND_AST_STMT_LIST，这个类型表示当前节点下有多个独立的节点，各child都是独立的语句生成的节点，所以依次编译即可，直到到达有效节点位置(非ZEND_AST_STMT_LIST节点)，然后调用 `zend_compile_stmt` 编译当前节点：

```
void zend_compile_stmt(zend_ast *ast)
{
    CG(zend_lineno) = ast->lineno;

    switch (ast->kind) {
        case xxx:
            ...
        break;
        case ZEND_AST_ECHO:
            zend_compile_echo(ast);
            break;
        ...
        default:
        {
            znode result;
            zend_compile_expr(&result, ast);
            zend_do_free(&result);
        }
    }

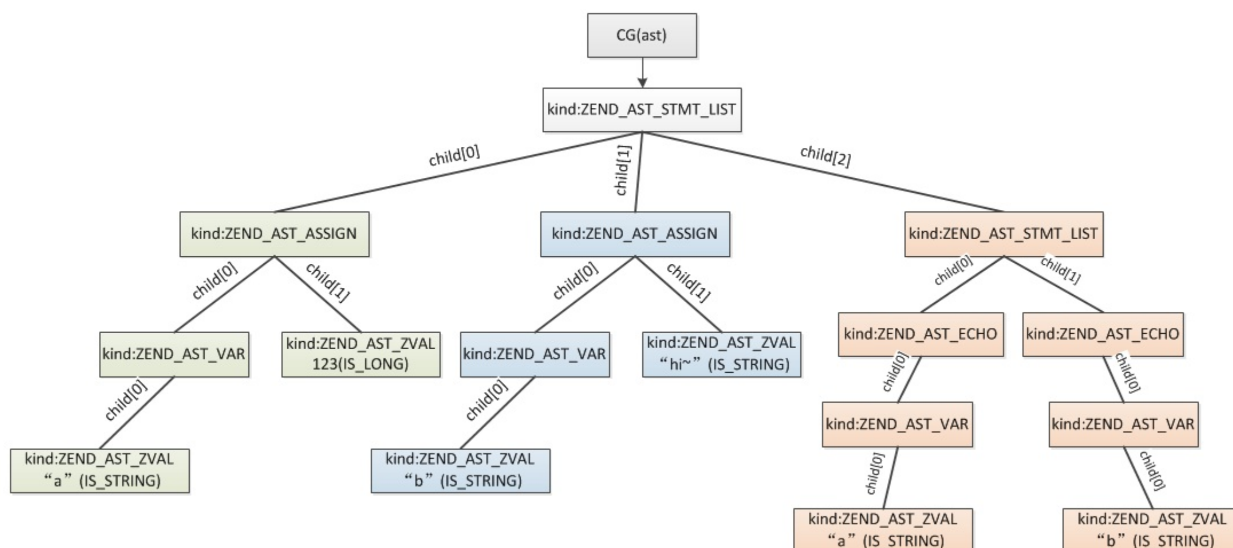
    if (FC(declarables).ticks && !zend_is_unticked_stmt(ast)) {
        zend_emit_tick();
    }
}
```

主要根据不同的节点类型(kind)作不同的处理，我们不会把每种类型的处理都讲一遍，这里还是根据上一节最后的例子挑几个看下具体的处理过程。

```
$a = 123;
$b = "hi~";

echo $a,$b;
```

zendparse()阶段生成的AST：



下面的过程比较复杂，有的函数会多次递归调用，我们根据例子一步步去看下，如果你对PHP各个语法实现比较熟悉再去看整个AST的编译过程就会比较轻松。

> **(1)**、首先从根节点开始，有3个child，第一个节点类型为ZEND_AST_ASSIGN，zend_compile_stmt()中走到default分支
>
> **(2)**、ZEND_AST_ASSIGN类型由zend_compile_expr()处理：

```
void zend_compile_expr(znode *result, zend_ast *ast)
{
    CG(zend_lineno) = zend_ast_get_lineno(ast);
    switch (ast->kind) {
        case ZEND_AST_ZVAL:
            ZVAL_COPY(&result->u.constant, zend_ast_get_zval(ast
));
            result->op_type = IS_CONST;
            return;
        case ZEND_AST_VAR:
            zend_compile_var(result, ast, BP_VAR_R);
            return;
        case ZEND_AST_ASSIGN:
            zend_compile_assign(result, ast);
            return;
        ...
    }
}
```

> 继续进入zend_compile_assign()： ```c void zend_compile_assign(znode *result, zend_ast* ast) { zend_ast *var_ast = ast->child[0]; //变量名 zend_ast* expr_ast = ast->child[1];//变量值表达式

```
znode var_node, expr_node;
zend_op *opline;
uint32_t offset;

if (is_this_fetch(var_ast)) { //检查变量名是否为this，变量名不能是thi
s
    zend_error_noreturn(E_COMPILE_ERROR, "Cannot re-assign $this
");
}

//比如这样写：my_function() = 123;即：将函数的返回值作为变量名将报错
zend_ensure_writable_variable(var_ast);

switch (var_ast->kind) {
    case ZEND_AST_VAR:
    case ZEND_AST_STATIC_PROP:
        offset = zend_delayed_compile_begin();
        zend_delayed_compile_var(&var_node, var_ast, BP_VAR_W);
//生成变量名的znode，这个结构只在这个地方临时用，所以直接分配在stack上
        zend_compile_expr(&expr_node, expr_ast); //递归编译变量值表
达式，最终需要得到一个ZEND_AST_ZVAL的节点
        zend_delayed_compile_end(offset);
        zend_emit_op(result, ZEND_ASSIGN, &var_node, &expr_node)
; //生成一条op
        return;
    ...
}
```

}

```
> 这个地方主要有三步关键操作：

>> __第1步：__ 变量赋值操作有两部分：变量名、变量值，所以首先是针对变量名的
操作，介绍zend_op_array时曾提到每个PHP变量都有一个编号，变量的读写都是根据
这个编号操作的，这个编号最早就是这一步生成的。
```

````
![](../img/zend_lookup_cv.png)

>> 中间过程我们不再细看，这里重点看下变量编号的过程，这个过程比较简单，每发
现一个变量就遍历zend_op_array.vars数组，看此变量是否已经保存，没有保存的
话则存入vars，然后后续变量的使用都是用的这个变量在数组中的下标，比如第一次定
义的时候：`$a = 123；`将$a编号为0，然后：`echo $a;`再次使用时会遍历vars
，发现已经存在，直接用其下标操作$a。
```c
static int lookup_cv(zend_op_array *op_array, zend_string* name)
{
    int i = 0;
    zend_ulong hash_value = zend_string_hash_val(name);

    //遍历op_array.vars检查此变量是否已存在
    while (i < op_array->last_var) {
        if (ZSTR_VAL(op_array->vars[i]) == ZSTR_VAL(name) ||
                (ZSTR_H(op_array->vars[i]) == hash_value &&
                 ZSTR_LEN(op_array->vars[i]) == ZSTR_LEN(name) &
&
                 memcmp(ZSTR_VAL(op_array->vars[i]), ZSTR_VAL(na
me), ZSTR_LEN(name)) == 0)) {
            zend_string_release(name);
            return (int)(zend_intptr_t)ZEND_CALL_VAR_NUM(NULL, i
);
        }
        i++;
    }
    //这是一个新变量
    i = op_array->last_var;
    op_array->last_var++;
    if (op_array->last_var > CG(context).vars_size) {
        CG(context).vars_size += 16; /* FIXME */
        op_array->vars = erealloc(op_array->vars, CG(context).va
rs_size * sizeof(zend_string*));//扩容vars
    }

    op_array->vars[i] = zend_new_interned_string(name);
    return (int)(zend_intptr_t)ZEND_CALL_VAR_NUM(NULL, i); //传N
ULL时返回的是96 + i*sizeof(zval)
````
````

```
}
```

> 注意：这里变量的编号从**0**、**1**、**2**、**3...**依次递增的，但是实际使用中并不是直接用的这个下标，而是转化成了内存偏移量**offset**，这个是 `ZEND_CALL_VAR_NUM` 宏处理的，所以变量偏移量实际是**96**、**112**、**128...**递增的，这个**96**是根据**zend_execute_data**大小设定的**(**不同的平台下对应的值可能不同**)**，下一篇介绍**zend**执行流程时会详细介绍这个结构。
> ```c
>
> # define ZEND_CALL_FRAME_SLOT \
>
> ```
> ((int)((ZEND_MM_ALIGNED_SIZE(sizeof(zend_execute_data)) +
>  ZEND_MM_ALIGNED_SIZE(sizeof(zval)) - 1) / ZEND_MM_ALIGNE
> D_SIZE(sizeof(zval))))
> ```

define ZEND_CALL_VAR_NUM(call, n) \

```
(((zval*)(call)) + (ZEND_CALL_FRAME_SLOT + ((int)(n))))
```

````
>> __第2步：__ 编译变量值表达式，再次调用zend_compile_expr()编译，示例中的情况比较简单，expr_ast.kind为ZEND_AST_ZVAL：
```c
void zend_compile_expr(znode *result, zend_ast *ast)
{
    switch (ast->kind) {
        case ZEND_AST_ZVAL:
            ZVAL_COPY(&result->u.constant, zend_ast_get_zval(ast)); //将变量值复制到znode.u.constant中
            result->op_type = IS_CONST; //类型为IS_CONST，这种value后面将会保存在zend_op_array.literals中
            return;
        ...
    }
}
````

> 第**3**步： 上面两步已经分别生成了变量赋值的op1、op2，下面就是根据这俩值生成opcode的过程。 ```c static zend_op *zend_emit_op(znode* result, zend_uchar opcode, znode *op1, znode* op2) { zend_op *opline = get_next_op(CG(active_op_array)); //当前zend_op_array下生成一条新的指令 opline->opcode = opcode;
````

```
//将op1、op2内容拷贝到zend_op中，设置op_type
//如果znode.op_type == IS_CONST，则会将znode.u.contstant值转移到zen
d_op_array.literals中
if (op1 == NULL) {
    SET_UNUSED(opline->op1);
} else {
    SET_NODE(opline->op1, op1);
}

if (op2 == NULL) {
    SET_UNUSED(opline->op2);
} else {
    SET_NODE(opline->op2, op2);
}

//如果此指令有返回值则想变量那样为返回值编号（后面分配局部变量时将根据这个编
号索引）
if (result) {
    zend_make_var_result(result, opline);
}
return opline;
```

}

static inline void zend_make_var_result(znode *result, zend_op* opline) { opline->result_type = IS_VAR; //返回值类型固定为IS_VAR opline->result.var = get_temporary_variable(CG(active_op_array)); //为返回值编个号，这个编号记在临时变量T上，上面介绍zend_op_array时说过T、last_var的区别 GET_NODE(result, opline->result); }

````
>> 到这我们示例中的第1条赋值语句就算编译完了，第2条同样是赋值，过程与上面相同，我们直接看最好一条输出的语句。

> __(3)、__ echo语句的编译:`echo $a,$b;`实际从编译后的语法树就可以看出，一次echo多个也被编译为多次echo了，所以示例中的用法与：`echo $a; echo $b;`等价，我们只分析其中一个就可以了。

![](../img/zend_ast_echo.png)

> `zend_compile_stmt()`中首先发现节点类型是`ZEND_AST_STMT_LIST`，然后调用`zend_compile_stmt_list()`分别编译child，具体的流程如下图所示：

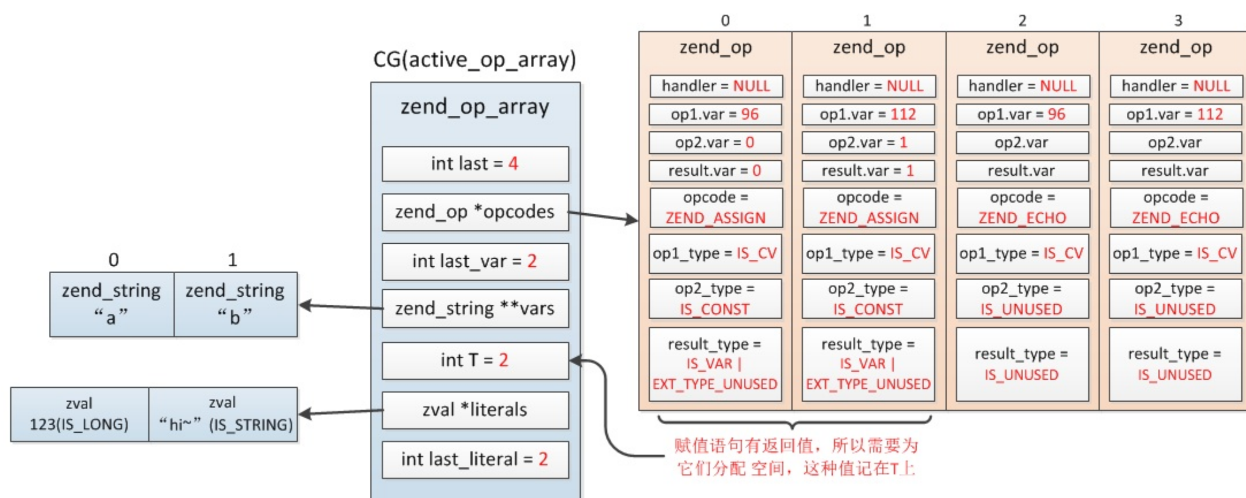
![](../img/zend_ast_echo_p.png)

> 最后生成`zend_op`的过程：
```c
void zend_compile_echo(zend_ast *ast)
{
    zend_op *opline;
    zend_ast *expr_ast = ast->child[0];

    znode expr_node;
    zend_compile_expr(&expr_node, expr_ast);

    opline = zend_emit_op(NULL, ZEND_ECHO, &expr_node, NULL);//生成1条新的opcode
    opline->extended_value = 0;
}
````

最终 `zend_compile_top_stmt()` 编译完成后整个编译流程基本是完成了， `CG(active_op_array)` 结构如下图所示，但是后面还有一个处理 `pass_two()` 。
````




```
ZEND_API int pass_two(zend_op_array *op_array)
{
    zend_op *opline, *end;

    if (!ZEND_USER_CODE(op_array->type)) {
        return 0;
    }

    //重置一些CG(context)的值，暂且忽略
    ...

    opline = op_array->opcodes;
    end = opline + op_array->last;
    while (opline < end) {
        switch(opline->opcode){
            //这里对一些操作进行针对性的处理，后面有遇到的情况我们再看
            ...
        }

        //如果是IS_CONST会将数组下标转化为内存偏移量，与IS_CV那种处理方
式相同
        //所以这里实际就是将0、1、2...转为为16、32、48...（即：编号*siz
eof(zval)）
        if (opline->op1_type == IS_CONST) {
            ZEND_PASS_TWO_UPDATE_CONSTANT(op_array, opline->op1)
;
        } else if (opline->op1_type & (IS_VAR|IS_TMP_VAR)) {
            //上面作相同的处理，不同的是这里的起始值是接着IS_CV的
```

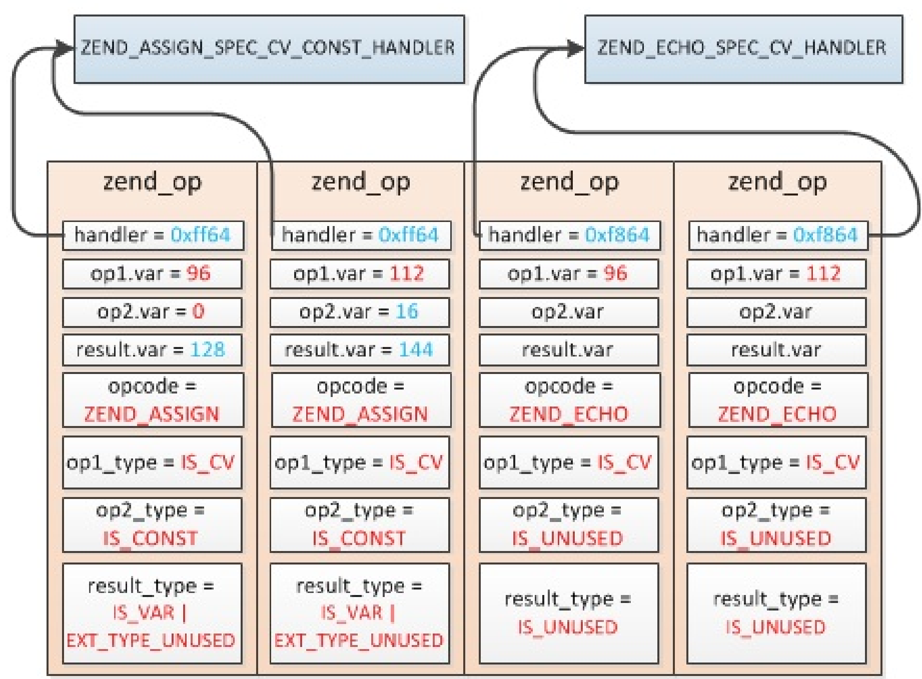
```
            opline->op1.var = (uint32_t)(zend_intptr_t)ZEND_CALL
_VAR_NUM(NULL, op_array->last_var + opline->op1.var);
        }
        //与op1完全相同
        if (opline->op2_type == IS_CONST) {
            ZEND_PASS_TWO_UPDATE_CONSTANT(op_array, opline->op2)
;
        } else if (opline->op2_type & (IS_VAR|IS_TMP_VAR)) {
            opline->op2.var = (uint32_t)(zend_intptr_t)ZEND_CALL
_VAR_NUM(NULL, op_array->last_var + opline->op2.var);
        }
        //返回值与op1/2相同处理
        if (opline->result_type & (IS_VAR|IS_TMP_VAR)) {
            opline->result.var = (uint32_t)(zend_intptr_t)ZEND_C
ALL_VAR_NUM(NULL, op_array->last_var + opline->result.var);
        }
        //设置此opcode的处理handler
        ZEND_VM_SET_OPCODE_HANDLER(opline);
        opline++;
    }

    //标识当前op_array已执行过此操作
    op_array->fn_flags |= ZEND_ACC_DONE_PASS_TWO;
    return 0;
}
```

抛开特殊opcode的处理，`pass_two()`主要有两个重要操作：

- (1)将IS_CONST、IS_VAR、IS_TMP_VAR类型的操作数、返回值转化为内存偏移量，与上面提到的IS_CV变量的处理一样，其中IS_CONST类型起始值为0，然后按照编号依次递增sizeof(zval)，而IS_VAR、IS_TMP_VAR唯一的不同时它的初始值接着IS_CV的，简单的讲就是先安排PHP变量的，然后接着才是各条语句的中间值、返回值
- (2)另外一个重要操作就是设置各指令的处理handler，这个前面《3.1.2.1.1 handler》已经介绍过其索引规则

经过`pass_two()`处理后opcodes的样子：



总结：

到这里整个PHP编译阶段就算全部完成了，最终编译的结果就是zend_op_array，其中最核心的操作就是AST的编译了，有兴趣的可以多写几个例子去看下不同节点类型的处理方式。

另外，编译阶段很关键的一个操作就是确定了各个 变量、中间值、临时值、返回值、字面量 的 内存编号 ，这个地方非常重要，后面介绍执行流程时也会用到。

3.2 函数实现

函数，通俗的讲就是一组操作的集合，给予特定的输入将对应特定的输出。

3.2.1 用户自定义函数的实现

用户自定义函数是指我们在PHP脚本通过function定义的函数：

```
function my_func(){
    ...
}
```

汇编中函数对应的是一组独立的汇编指令，然后通过call指令实现函数的调用。前面已经说过PHP编译的结果是opcode数组，与汇编指令对应。PHP用户自定义函数的实现就是将函数编译为独立的opcode数组，调用时分配独立的执行栈依次执行opcode，所以自定义函数对于zend而言并没有什么特别之处，只是将opcode进行了打包封装。PHP脚本中函数之外的指令,整个可以认为是一个函数(或者理解为main函数更直观)。

```
/* function main(){ */

$a = 123;
echo $a;

/* } */
```

3.2.1.1 函数的存储结构

下面具体看下PHP中函数的结构：

```
typedef union  _zend_function        zend_function;

//zend_compile.h
union _zend_function {
    zend_uchar type;    /* MUST be the first element of this str
uct! */

    struct {
        zend_uchar type;  /* never used */
        zend_uchar arg_flags[3]; /* bitset of arg_info.pass_by_r
eference */
        uint32_t fn_flags;
        zend_string *function_name;
        zend_class_entry *scope; //成员方法所属类，面向对象实现中用到
        union _zend_function *prototype;
        uint32_t num_args; //参数数量
        uint32_t required_num_args; //必传参数数量
        zend_arg_info *arg_info; //参数信息
    } common;

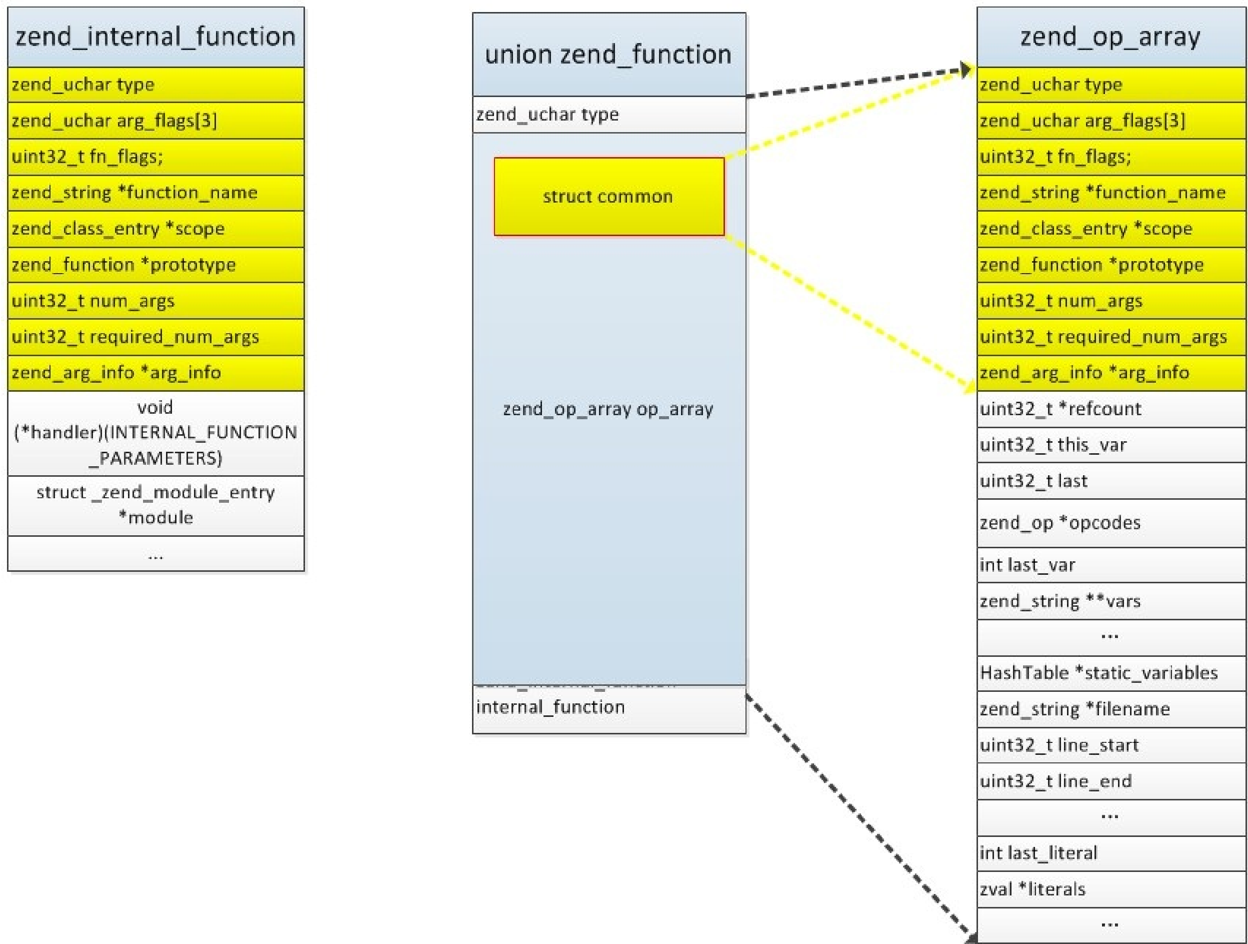
    zend_op_array op_array; //函数实际编译为普通的zend_op_array
    zend_internal_function internal_function;
};
```

这是一个union，因为PHP中函数除了用户自定义函数还有一种：内部函数，内部函数是通过扩展或者内核提供的C函数，比如time、array系列等等，内部函数稍后再作分析。

内部函数主要用到 `internal_function` ，而用户自定义函数编译完就是一个普通的opcode数组，用的是 `op_array` （注意：op_array、internal_function是嵌入的两个结构，而不是一个单独的指针），除了这两个上面还有一个 `type` 跟 `common` ，这俩是做什么用的呢？

经过比较发现 `zend_op_array` 与 `zend_internal_function` 结构的起始位置都有 `common` 中的几个成员，如果你对C的内存比较了解应该会马上想到它们的用法，实际 `common` 可以看作是 `op_array` 、 `internal_function` 的header，不管是什么哪种函数都可以通过 `zend_function.common.xx` 快速访问到 `zend_function.zend_op_array.xx` 及 `zend_function.zend_internal_fun`

`ction.xx` ，下面几个， `type` 同理，可以直接通过 `zend_function.type` 取到 `zend_function.op_array.type` 及 `zend_function.internal_function.type` 。



函数是在编译阶段确定的，那么它们存在哪呢？

在PHP脚本的生命周期中有一个非常重要的值 `executor_globals` (非ZTS下)，类型是 `struct _zend_executor_globals` ，它记录着PHP生命周期中所有的数据，如果你写过PHP扩展一定用到过 `EG` 这个宏，这个宏实际就是对 `executor_globals` 的操作: `define EG(v) (executor_globals.v)`

`EG(function_table)` 是一个哈希表，记录的就是PHP中所有的函数。

PHP在编译阶段将用户自定义的函数编译为独立的opcodes，保存在 `EG(function_table)` 中，调用时重新分配新的zend_execute_data(相当于运行栈)，然后执行函数的opcodes，调用完再还原到旧的 `zend_execute_data` ，继续执行，关于zend引擎execute阶段后面会详细分析。

3.2.1.2 函数参数

函数参数在内核实现上与函数内的局部变量实际是一样的，上一篇我们介绍编译的时候提供局部变量会有一个单独的 编号 ，而函数的参数与之相同，参数名称也在zend_op_array.vars中，编号首先是从参数开始的，所以按照参数顺序其编号依次为0、1、2...(转化为相对内存偏移量就是96、112、128...)，然后函数调用时首先会在调用位置将参数的value复制到各参数各自的位置，详细的传参过程我们在执行一篇再作说明。

比如：

```
function my_function($a, $b = "aa"){
    $ret = $a . $b;
    return $ret;
}
```

编译完后各变量的内存偏移量编号：

```
$a   => 96
$b   => 112
$ret => 128
```

与下面这么写一样：

```
function my_function(){
    $a = NULL;
    $b = "aa";
    $ret = $a . $b;
    return $ret;
}
```

另外参数还有其它的信息，这些信息通过 `zend_arg_info` 结构记录：

```c
typedef struct _zend_arg_info {
    zend_string *name; //参数名
    zend_string *class_name;
    zend_uchar type_hint; //显式声明的参数类型，比如(array $param_1)
    zend_uchar pass_by_reference; //是否引用传参，参数前加&的这个值就是1
    zend_bool allow_null; //是否允许为NULL,注意：这个值并不是用来表示参数是否为必传的
    zend_bool is_variadic; //是否为可变参数，即...用法，与golang的用法相同，5.6以上新增的一个用法：function my_func($a, ...$b){...}
} zend_arg_info;
```

每个参数都有一个上面的结构，所有参数的结构保存在 `zend_op_array.arg_info` 数组中，这里有一个地方需要注意： `zend_op_array->arg_info` 数组保存的并不全是输入参数，如果函数声明了返回值类型则也会为它创建一个 `zend_arg_info` ，这个结构在arg_info数组的第一个位置，这种情况下 `zend_op_array->arg_info` 指向的实际是数组的第二个位置，返回值的结构通过 `zend_op_array->arg_info[-1]` 读取，这里先单独看下编译时的处理：

```
//函数参数的编译
void zend_compile_params(zend_ast *ast, zend_ast *return_type_as
t)
{
    zend_ast_list *list = zend_ast_get_list(ast);
    uint32_t i;
    zend_op_array *op_array = CG(active_op_array);
    zend_arg_info *arg_infos;

    if (return_type_ast) {
        //声明了返回值类型：function my_func():array{...}
        //多分配一个zend_arg_info
        arg_infos = safe_emalloc(sizeof(zend_arg_info), list->ch
ildren + 1, 0);
        ...
        arg_infos->allow_null = 0;
        ...
        //arg_infos指向了下一个位置
        arg_infos++;
        op_array->fn_flags |= ZEND_ACC_HAS_RETURN_TYPE;
    } else {
        //没有声明返回值类型
        if (list->children == 0) {
            return;
        }
        arg_infos = safe_emalloc(sizeof(zend_arg_info), list->ch
ildren, 0);
    }
    ...

    op_array->num_args = list->children;
    //声明了返回值的情况下arg_infos已经指向了数组的第二个元素
    op_array->arg_info = arg_infos;
}
```

3.2.1.3 函数的编译

我们在上一篇文章介绍过PHP代码的编译过程，主要是PHP->AST->Opcodes的转化，上面也说了函数其实就是将一组PHP代码编译为单独的opcodes，函数的调用就是不同opcodes间的切换，所以函数的编译过程与普通PHP代码基本一致，只是会有一些特殊操作，我们以3.2.1.2开始那个例子简单看下编译过程。

普通函数的语法解析规则：

```
function_declaration_statement:
    function returns_ref T_STRING backup_doc_comment '(' paramet
er_list ')' return_type
    '{' inner_statement_list '}'
        { $$ = zend_ast_create_decl(ZEND_AST_FUNC_DECL, $2, $1, $
4,
            zend_ast_get_str($3), $6, NULL, $10, $8); }
;
```

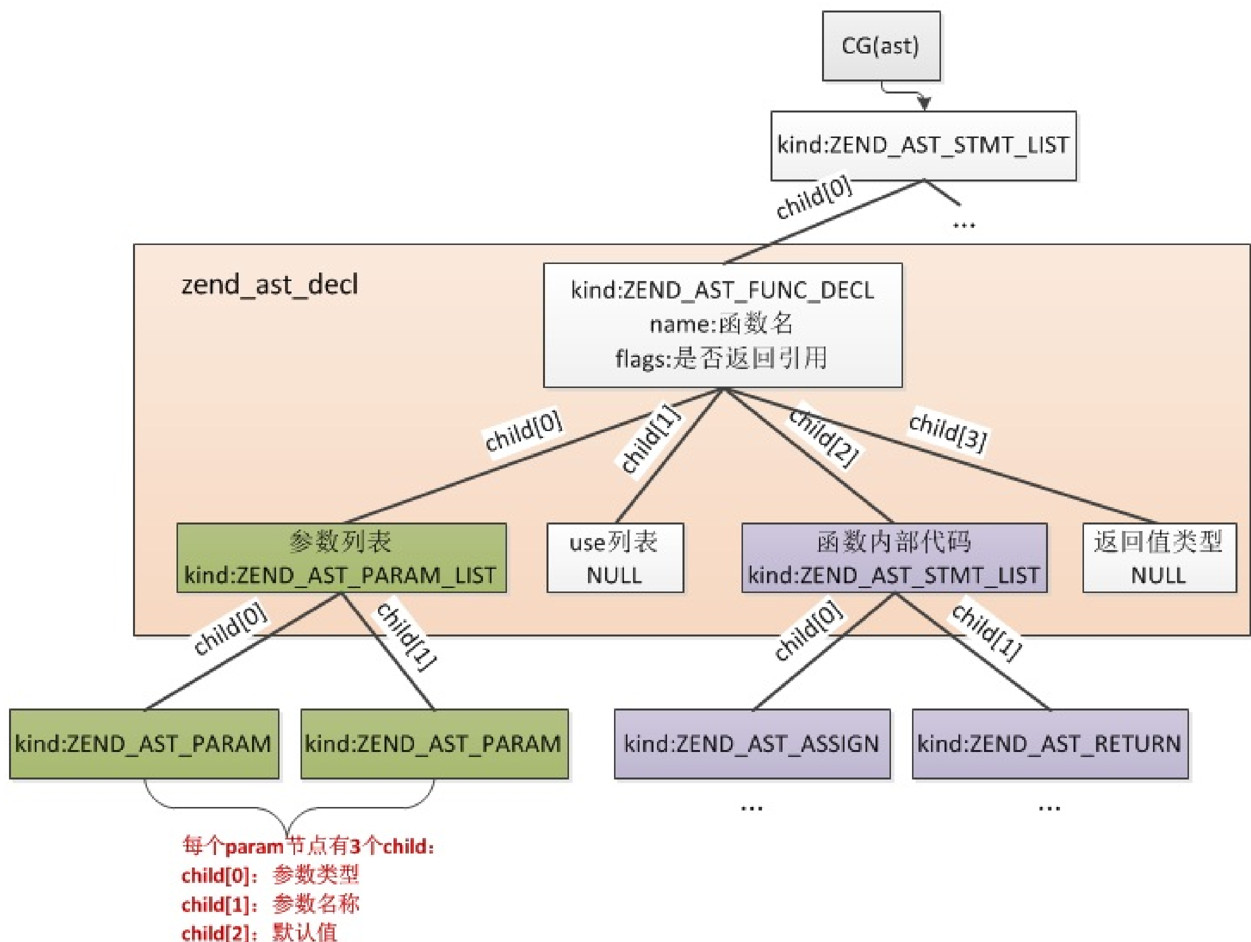
规则主要由五部分组成：

- **returns_ref:** 是否返回引用，在函数名前加&，比如function &test(){...}
- **T_STRING:** 函数名
- **parameter_list:** 参数列表
- **return_type:** 返回值类型
- **inner_statement_list:** 函数内部代码

函数生成的抽象语法树根节点类型是zend_ast_decl，所有函数相关的信息都记录在这个节点中(除了函数外类也是用的这个)：

```
typedef struct _zend_ast_decl {
    zend_ast_kind kind; //函数就是ZEND_AST_FUNC_DECL，类则是ZEND_AST_CLASS
    zend_ast_attr attr; /* Unused - for structure compatibility */
    uint32_t start_lineno; //函数起始行
    uint32_t end_lineno;  //函数结束行
    uint32_t flags;   //其中一个标识位用来标识返回值是否为引用，是则为ZEND_ACC_RETURN_REFERENCE
    unsigned char *lex_pos;
    zend_string *doc_comment;
    zend_string *name;  //函数名
    zend_ast *child[4]; //child有4个子节点，分别是：参数列表节点、use列表节点、函数内部表达式节点、返回值类型节点
} zend_ast_decl;
```

上面的例子最终生成的语法树：



具体编译为opcodes的过程在 `zend_compile_func_decl()` 中：

```
void zend_compile_func_decl(znode *result, zend_ast *ast)
{
    zend_ast_decl *decl = (zend_ast_decl *) ast;
    zend_ast *params_ast = decl->child[0]; //参数列表
    zend_ast *uses_ast = decl->child[1]; //use列表
    zend_ast *stmt_ast = decl->child[2]; //函数内部
    zend_ast *return_type_ast = decl->child[3]; //返回值类型
    zend_bool is_method = decl->kind == ZEND_AST_METHOD; //是否为成员函数

    //这里保存当前正在编译的zend_op_array：CG(active_op_array)，然后重新为函数生成一个新的zend_op_array，
    //函数编译完再将旧的还原
    zend_op_array *orig_op_array = CG(active_op_array);
    zend_op_array *op_array = zend_arena_alloc(&CG(arena), sizeof(zend_op_array)); //新分配zend_op_array
    ...

    if (is_method) {
        zend_bool has_body = stmt_ast != NULL;
        zend_begin_method_decl(op_array, decl->name, has_body);
    } else {
        zend_begin_func_decl(result, op_array, decl); //注意这里会在当前zend_op_array（不是新生成的函数那个）生成一条ZEND_DECLARE_FUNCTION的opcode
    }
    CG(active_op_array) = op_array;
    ...

    zend_compile_params(params_ast, return_type_ast); //编译参数
    if (uses_ast) {
        zend_compile_closure_uses(uses_ast);
    }
    zend_compile_stmt(stmt_ast); //编译函数内部语法
    ...
    pass_two(CG(active_op_array));
    ...
    CG(active_op_array) = orig_op_array; //还原之前的
}
```

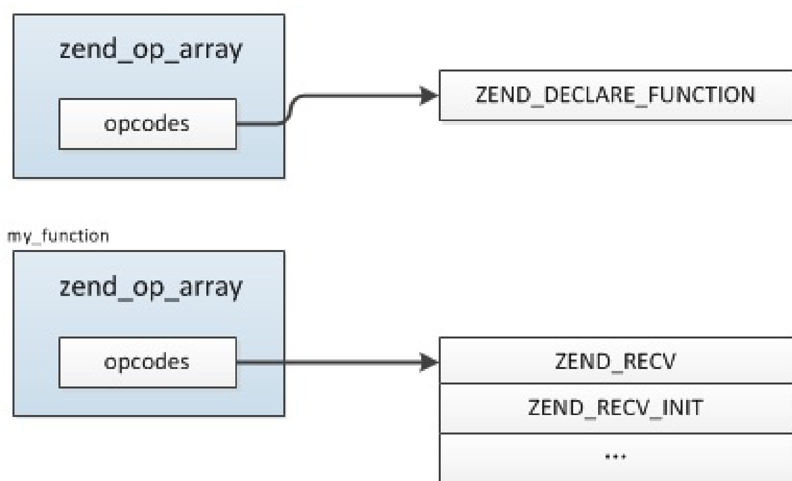
> 编译过程主要有这么几个处理：
>
> **(1)** 保存当前正在编译的zendop*array*，新分配一个结构，因为每个函数、*include*的文件都对应独立的一个*zend_op_array*，通过*CG(active_op_array)*记录当前编译所属*zend_op_array*，所以开始编译函数时就需要将这个值保存下来，等到函数编译完成再还原回去；另外还有一个关键操作：`zend_begin_func_decl`，这里会在当前*zend_op_array*（不是新生成的函数那个）生成一条 __*ZEND_DECLARE_FUNCTION* 的opcode，也就是函数声明操作。

```
$a = 123;  //当前为CG(active_op_array) = zend_op_array_1，编译到这时此opcode加到zend_op_array_1

//新分配一个zend_op_array_2，并将当前CG(active_op_array)保存到origin_op_array，
//然后将CG(active_op_array)=zend_op_array_2
function test(){
    $b = 234; //编译到zend_op_array_2
}//函数编译结束，将CG(active_op_array) = origin_op_array，切回zend_op_array_1
$c = 345; //编译到zend_op_array_1
```

> (2) 编译参数列表，函数的参数我们在上一小节已经介绍，完整的参数会有三个组成：参数类型(可选)、参数名、默认值(可选)，这三部分分别保存在参数节点的三个child节点中，编译参数的过程有两个关键操作：
>
> > 操作1： 为每个参数编号
> >
> > 操作2： 每个参数生成一条opcode，如果是可变参数其opcode=ZEND_RECV_VARIADIC，如果有默认值则为ZEND_RECV_INIT，否则为ZEND_RECV
>
> 上面的例子中$a编号为96，$b为112，同时生成了两条opcode：ZEND_RECV、ZEND_RECV_INIT，调用的时候会根据具体传参数量跳过部分opcode，比如这个函数我们这么调用 `my_function($a)` 则ZEND_RECV这条opcode就直接跳过了，然后执行ZEND_RECV_INIT将默认值写到112位置，具体的编译过程在 `zend_compile_params()` 中，上面已经介绍过。
>
> 参数默认值的保存与普通变量赋值相同： `$a = array()` ， `array()` 保存在literals，参数的默认值也是如此。
>
> (3) 编译函数内部语法，这个跟普通PHP代码编译过程无异。
>
> (4) pass_two()，上一篇介绍过，不再赘述。

最终生成两个zend_op_array：



总体来看，PHP在逐行编译时发现一个function则生成一条ZEND_DECLARE_FUNCTION的opcode，然后调到函数中编译函数，编译完再跳回去继续下面的编译，这里多次提到ZEND_DECLARE_FUNCTION这个opcode是

因为在函数编译结束后还有一个重要操作：`zend_do_early_binding()`，前面我们说过总的编译入口在`zend_compile_top_stmt()`，这里会对每条语法逐条编译，而函数、类在编译完成后还有后续的操作：

```
void zend_compile_top_stmt(zend_ast *ast)
{
    ...
    if (ast->kind == ZEND_AST_STMT_LIST) {
        for (i = 0; i < list->children; ++i) {
            zend_compile_top_stmt(list->child[i]);
        }
    }

    zend_compile_stmt(ast); //编译各条语法，函数也是在这里编译完成

    //函数编译完成后
    if (ast->kind == ZEND_AST_FUNC_DECL || ast->kind == ZEND_AST
_CLASS) {
        CG(zend_lineno) = ((zend_ast_decl *) ast)->end_lineno;
        zend_do_early_binding();
    }
}
```

`zend_do_early_binding()`核心工作就是 将**function**、**class**加到**CG(function_table)**、**CG(class_table)**中，加入成功了就直接把**ZEND_DECLARE_FUNCTION** 这条opcode干掉了，加入失败的话则保留，这个相当于 有一部分**opcode**在『编译时』提前执行了，这也是为什么PHP中可以先调用函数再声明函数的原因，比如：

```
$a = 1234;

echo my_function($a);

function my_function($a){
    ...
}
```

实际原始的opcode以及执行顺序：



类的情况也是如此，后面我们再作说明。

3.2.1.4 匿名函数

匿名函数（Anonymous functions），也叫闭包函数（closures），允许临时创建一个没有指定名称的函数。最经常用作回调函数（callback）参数的值。当然，也有其它应用的情况。

官网的示例：

```
$greet = function($name)
{
    printf("Hello %s\r\n", $name);
};

$greet('World');
$greet('PHP');
```

这里提匿名函数只是想说明编译函数时那个use的用法:

匿名函数可以从父作用域中继承变量。 任何此类变量都应该用 **use** 语言结构传递进去。

```
$message = 'hello';

$example = function () use ($message) {
    var_dump($message);
};
$example();
```

3.2.2 内部函数

上一节已经提过，内部函数指的是由内核、扩展提供的C语言编写的function，这类函数不需要经历opcode的编译过程，所以效率上要高于PHP用户自定义的函数，调用时与普通的C程序没有差异。

Zend引擎中定义了很多内部函数供用户在PHP中使用，比如：define、defined、strlen、method_exists、class_exists、function_exists......等等，除了Zend引擎中定义的内部函数，PHP扩展中也提供了大量内部函数，我们也可以灵活的通过扩展自行定制。

3.2.2.1 内部函数结构

上一节介绍 `zend_function` 为union，其中 `internal_function` 就是内部函数用到的，具体结构：

```
//zend_complie.h
typedef struct _zend_internal_function {
    /* Common elements */
    zend_uchar type;
    zend_uchar arg_flags[3]; /* bitset of arg_info.pass_by_refer
ence */
    uint32_t fn_flags;
    zend_string* function_name;
    zend_class_entry *scope;
    zend_function *prototype;
    uint32_t num_args;
    uint32_t required_num_args;
    zend_internal_arg_info *arg_info;
    /* END of common elements */

    void (*handler)(INTERNAL_FUNCTION_PARAMETERS); //函数指针，展
开：void (*handler)(zend_execute_data *execute_data, zval *return
_value)
    struct _zend_module_entry *module;
    void *reserved[ZEND_MAX_RESERVED_RESOURCES];
} zend_internal_function;
```

`zend_internal_function` 头部是一个与 `zend_op_array` 完全相同的common结构。

下面看下如何定义一个内部函数。

3.2.2.2 定义与注册

内部函数与用户自定义函数冲突，用户无法在PHP代码中覆盖内部函数，执行PHP脚本时会提示error错误。

内部函数的定义非常简单，我们只需要创建一个普通的C函数，然后创建一个 `zend_internal_function` 结构添加到 **EG(function_table)** (也可能是CG(function*table),取决于在哪一阶段注册)中即可使用，内部函数*通常 情况下是在php_module_startup阶段注册的，这里之所以说通常是按照标准的扩展定义，除了扩展提供的方式我们可以在任何阶段自由定义内部函数，当然并不建议这样做。下面我们先不讨论扩展标准的定义方式，我们先自己尝试下如何注册一个内部函数。

根据 `zend_internal_function` 的结构我们知道需要定义一个handler：

```
void qp_test(INTERNAL_FUNCTION_PARAMETERS)
{
    printf("call internal function 'qp_test'\n");
}
```

然后创建一个内部函数结构(我们在扩展PHP_MINIT_FUNCTION方法中注册，也可以在其他位置)：

```
PHP_MINIT_FUNCTION(xxxxxx)
{
    zend_string *lowercase_name;
    zend_function *reg_function;

    //函数名转小写，因为php的函数不区分大小写
    lowercase_name = zend_string_alloc(7, 1);
    zend_str_tolower_copy(ZSTR_VAL(lowercase_name), "qp_test", 7
);
    lowercase_name = zend_new_interned_string(lowercase_name);

    reg_function = malloc(sizeof(zend_internal_function));
    reg_function->internal_function.type = ZEND_INTERNAL_FUNCTIO
N; //定义类型为内部函数
    reg_function->internal_function.function_name = lowercase_na
me;
    reg_function->internal_function.handler = qp_test;

    zend_hash_add_ptr(CG(function_table), lowercase_name, reg_fu
nction); //注册到CG(function_table)符号表中
}
```

接着编译、安装扩展，测试：

```
qp_test();
```

结果输出： `call internal function 'qp_test'`

这样一个内部函数就定义完成了。这里有一个地方需要注意的我们把这个函数注册到 **CG(function_table)** 中去了，而不是 **EG(function_table)** ，这是因为在 `php_request_startup` 阶段会把 **CG(function_table)** 赋值给 **EG(function_table)** 。

上面的过程看着比较简单，但是在实际应用中不要这样做，PHP提供给我们一套标准的定义方式，接下来看下如何在扩展中按照官方方式提供一个内部函数。

首先也是定义C函数，这个通过 `PHP_FUNCTION` 宏定义：

```
PHP_FUNCTION(qp_test)
{
    printf("call internal function 'qp_test'\n");
}
```

然后是注册过程，这个只需要我们将所有的函数数组添加到扩展结构 `zend_module_entry.functions` 即可，扩展加载过程中会自动进行函数注册(见1.2节)，不需要我们干预：

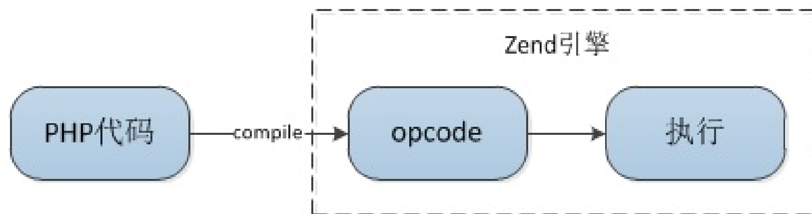
```
const zend_function_entry xxxx_functions[] = {
        PHP_FE(qp_test,   NULL)
        PHP_FE_END
};

zend_module_entry xxxx_module_entry = {
    STANDARD_MODULE_HEADER,
    "扩展名称",
    xxxx_functions,
    PHP_MINIT(timeout),
    PHP_MSHUTDOWN(timeout),
    PHP_RINIT(timeout),     /* Replace with NULL if there's nothing to do at request start */
    PHP_RSHUTDOWN(timeout), /* Replace with NULL if there's nothing to do at request end */
    PHP_MINFO(timeout),
    PHP_TIMEOUT_VERSION,
    STANDARD_MODULE_PROPERTIES
};
```

关于更多扩展中函数相关的用法会在后面扩展开发一章中详细介绍，这里不再展开。

3.3 Zend引擎执行过程

Zend引擎主要包含两个核心部分：编译、执行：



前面分析了Zend的编译过程以及PHP用户函数的实现，接下来分析下Zend引擎的执行过程。

3.3.1 数据结构

执行流程中有几个重要的数据结构，先看下这几个结构。

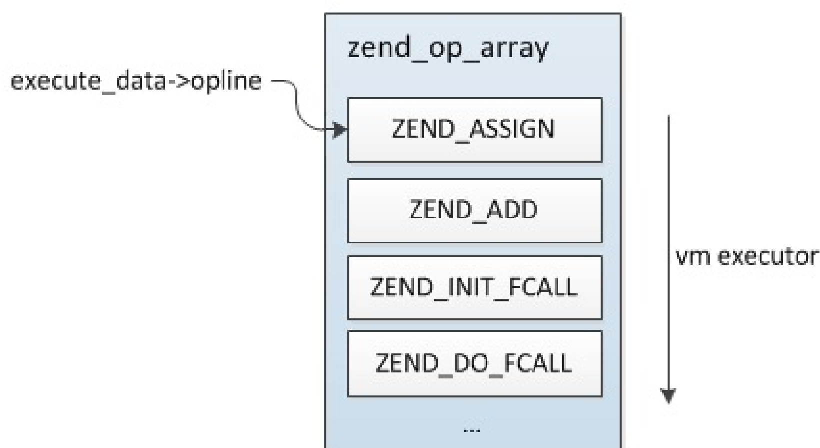
3.3.1.1 opcode

opcode是将PHP代码编译产生的Zend虚拟机可识别的指令，php7共有173个opcode，定义在 `zend_vm_opcodes.h` 中，PHP中的所有语法实现都是由这些opcode组成的。

```
struct _zend_op {
    const void *handler; //对应执行的C语言function，即每条opcode都有一个C function处理
    znode_op op1;   //操作数1
    znode_op op2;   //操作数2
    znode_op result; //返回值
    uint32_t extended_value;
    uint32_t lineno;
    zend_uchar opcode;  //opcode指令
    zend_uchar op1_type; //操作数1类型
    zend_uchar op2_type; //操作数2类型
    zend_uchar result_type; //返回值类型
};
```

3.3.1.2 zend_op_array

`zend_op_array` 是Zend引擎执行阶段的输入，整个执行阶段的操作都是围绕着这个结构，关于其具体结构前面我们已经讲过了。



这里再重复说下zend_op_array几个核心组成部分：

- **opcode**指令：即PHP代码具体对应的处理动作，与二进制程序中的代码段对应
- 字面量存储：PHP代码中定义的一些变量初始值、调用的函数名称、类名称、常量名称等等称之为字面量，这些值用于执行时初始化变量、函数调用等等
- 变量分配情况：与字面量类似，这里指的是当前opcodes定义了多少变量、临时变量，每个变量都有一个对应的编号，执行初始化按照总的数目一次性分配zval，使用时也完全按照编号索引，而不是根据变量名索引

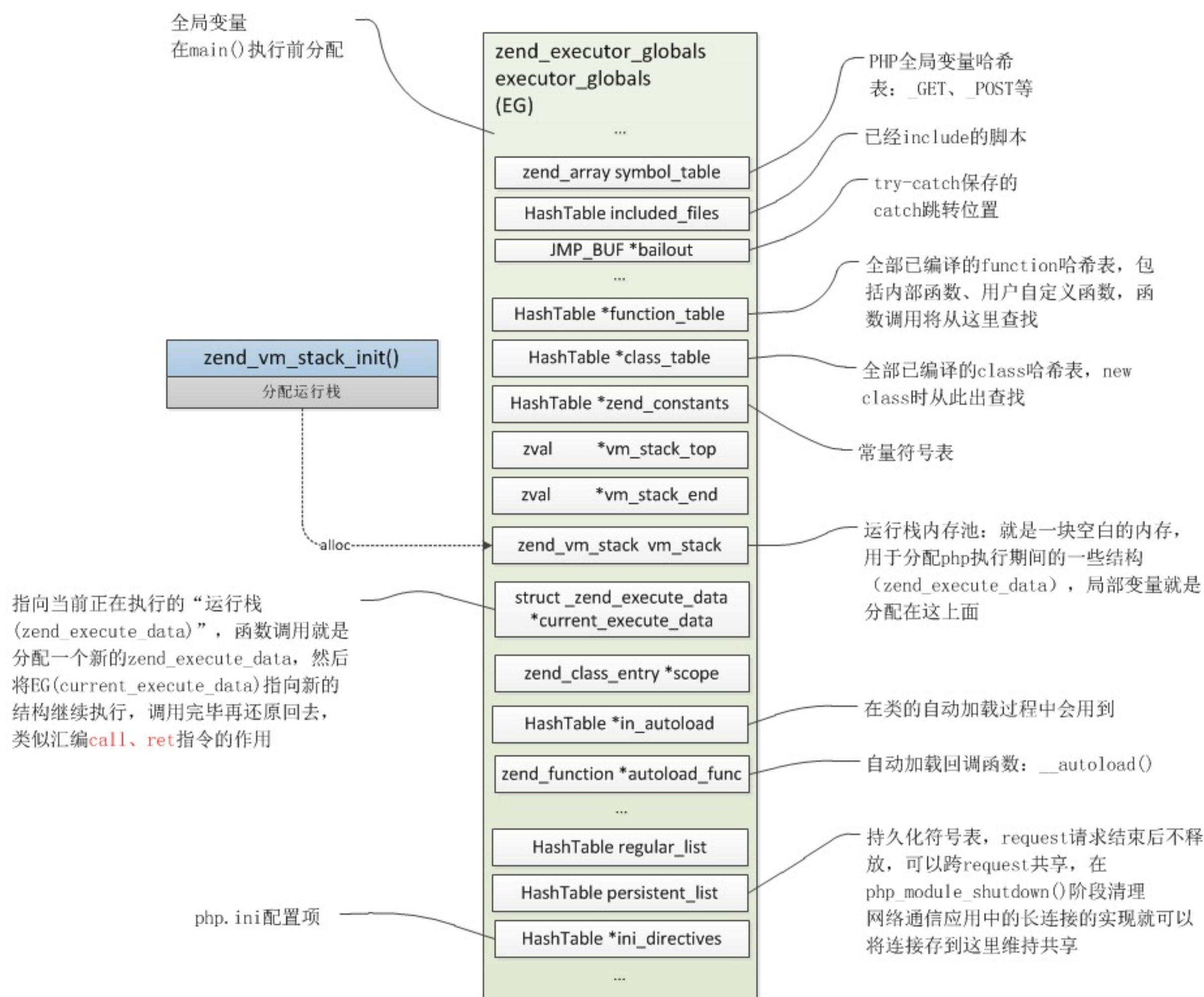
3.3.1.3 zend_executor_globals

`zend_executor_globals executor_globals` 是PHP整个生命周期中最主要的一个结构，是一个全局变量，在main执行前分配(非ZTS下)，直到PHP退出，它记录着当前请求全部的信息，经常见到的一个宏 `EG` 操作的就是这个结构。

```
//zend_compile.c
#ifndef ZTS
ZEND_API zend_compiler_globals compiler_globals;
ZEND_API zend_executor_globals executor_globals;
#endif

//zend_globals_macros.h
# define EG(v) (executor_globals.v)
```

`zend_executor_globals` 结构非常大，定义在 `zend_globals.h` 中，比较重要的几个字段含义如下图所示：



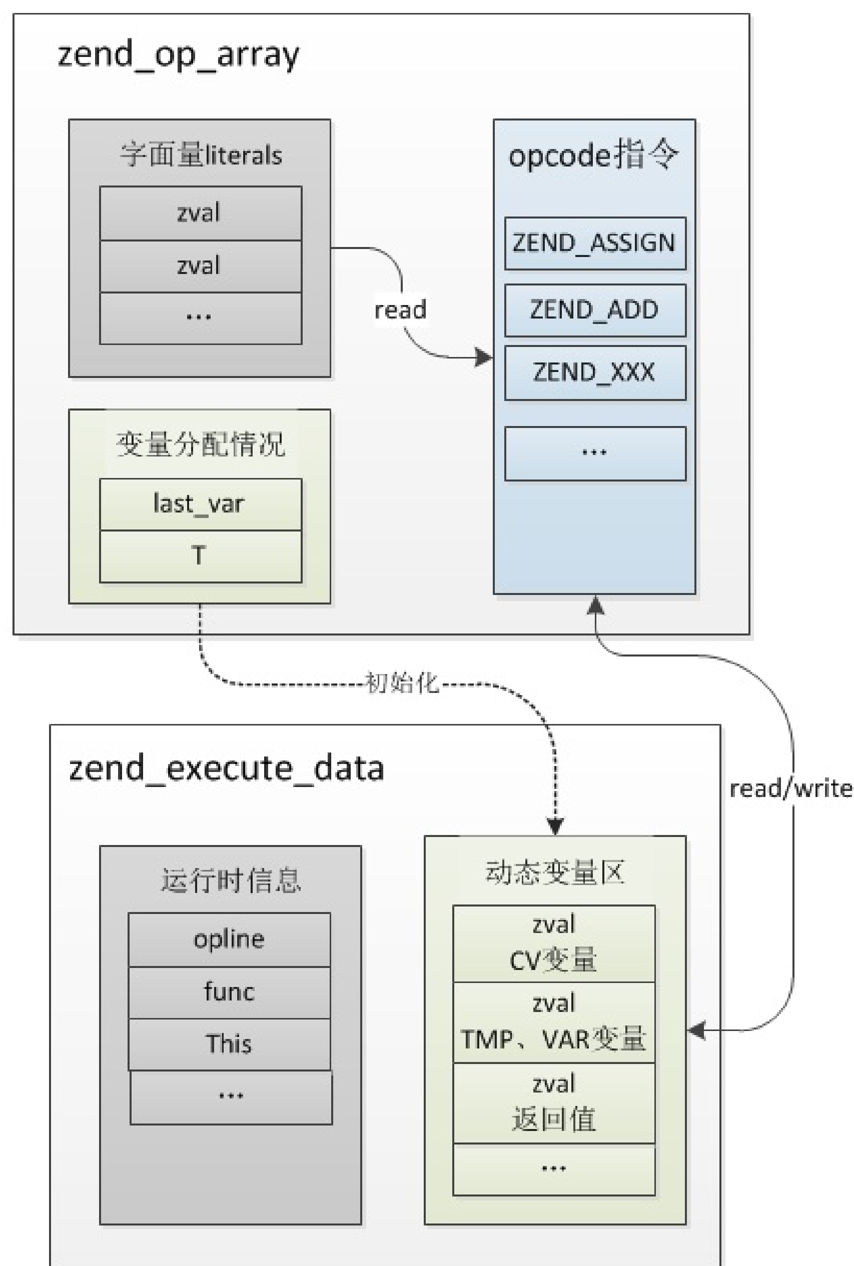
3.3.1.4 zend_execute_data

`zend_execute_data` 是执行过程中最核心的一个结构，每次函数的调用、include/require、eval等都会生成一个新的结构，它表示当前的作用域、代码的执行位置以及局部变量的分配等等，等同于机器码执行过程中stack的角色，后面分析具体执行流程的时候会详细分析其作用。

```
#define EX(element)             ((execute_data)->element)

//zend_compile.h
struct _zend_execute_data {
    const zend_op       *opline;  //指向当前执行的opcode，初始时指向zend_op_array起始位置
    zend_execute_data   *call;             /* current call */
    zval                *return_value;  //返回值指针
    zend_function       *func;          //当前执行的函数（非函数调用时为空）
    zval                 This;          //这个值并不仅仅是面向对象的this，还有另外两个值也通过这个记录：call_info + num_args，分别存在zval.u1.reserved、zval.u2.num_args
    zend_class_entry    *called_scope;  //当前call的类
    zend_execute_data   *prev_execute_data; //函数调用时指向调用位置作用空间
    zend_array          *symbol_table; //全局变量符号表
#if ZEND_EX_USE_RUN_TIME_CACHE
    void               **run_time_cache;   /* cache op_array->run_time_cache */
#endif
#if ZEND_EX_USE_LITERALS
    zval                *literals;  //字面量数组，与func.op_array->literals相同
#endif
};
```

zend_execute_data与zend_op_array的关联关系：



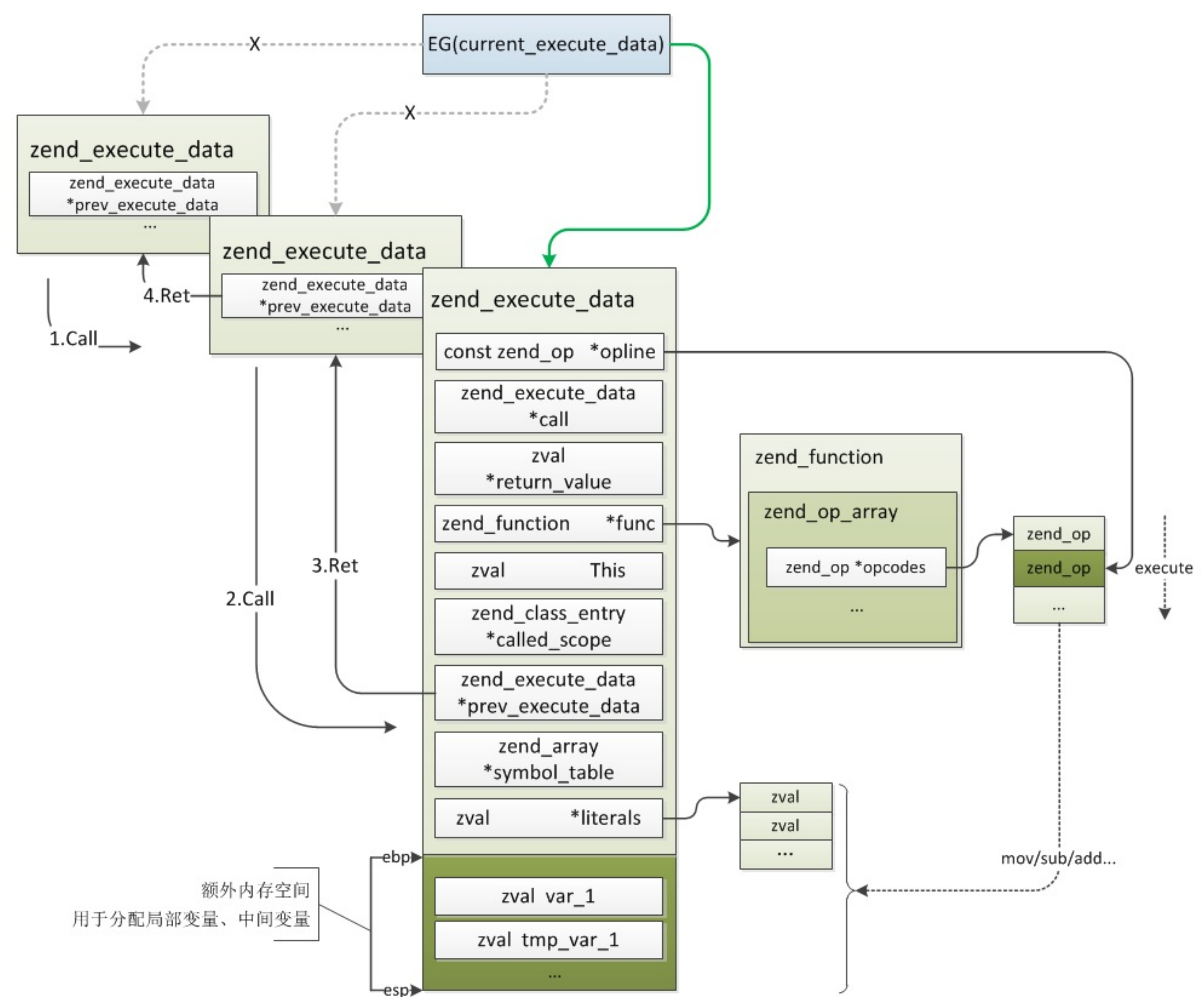
3.3.2 执行流程

Zend的executor与linux二进制程序执行的过程是非常类似的，在C程序执行时有两个寄存器ebp、esp分别指向当前作用栈的栈顶、栈底，局部变量全部分配在当前栈，函数调用、返回通过 `call` 、 `ret` 指令完成，调用时 `call` 将当前执行位置压入栈中，返回时 `ret` 将之前执行位置出栈，跳回旧的位置继续执行，在Zend VM中 `zend_execute_data` 就扮演了这两个角色， `zend_execute_data.prev_execute_data` 保存的是调用方的信息，实现了 `call/ret` ， `zend_execute_data` 后面会分配额外的内存空间用于局部变量的存储，实现了 `ebp/esp` 的作用。

注意：在执行前分配内存时并不仅仅是分配了 `zend_execute_data` 大小的空间，除了 `sizeof(zend_execute_data)` 外还会额外申请一块空间，用于分配局部变量、临时(中间)变量等，具体的分配过程下面会讲到。

Zend执行opcode的简略过程：

- **step1:** 为当前作用域分配一块内存，充当运行栈，zend_execute_data结构、所有局部变量、中间变量等等都在此内存上分配
- **step2:** 初始化全局变量符号表，然后将全局执行位置指针EG(current_execute_data)指向step1新分配的zend_execute_data，然后将zend_execute_data.opline指向op_array的起始位置
- **step3:** 从EX(opline)开始调用各opcode的C处理handler(即_zend_op.handler)，每执行完一条opcode将 `EX(opline)++` 继续执行下一条，直到执行完全部opcode，函数/类成员方法调用、if的执行过程：
    - **step3.1:** if语句将根据条件的成立与否决定 `EX(opline) + offset` 所加的偏移量，实现跳转
    - **step3.2:** 如果是函数调用，则首先从EG(function_table)中根据function_name取出此function对应的编译完成的zend_op_array，然后像step1一样新分配一个zend_execute_data结构，将EG(current_execute_data)赋值给新结构的 `prev_execute_data` ，再将EG(current_execute_data)指向新的zend_execute_data，最后从新的 `zend_execute_data.opline` 开始执行，切换到函数内部，函数执行完以后将EG(current_execute_data)重新指向EX(prev_execute_data)，释放分配的运行栈，销毁局部变量，继续从原来函数调用的位置执行
    - **step3.3:** 类方法的调用与函数基本相同，后面分析对象实现的时候再详细分析
- **step4:** 全部opcode执行完成后将step1分配的内存释放，这个过程会将所有的局部变量"销毁"，执行阶段结束



接下来详细看下整个流程。

Zend执行入口为位于 `zend_vm_execute.h` 文件中的**zend_execute()**：

```
ZEND_API void zend_execute(zend_op_array *op_array, zval *return
_value)
{
    zend_execute_data *execute_data;

    if (EG(exception) != NULL) {
        return;
    }

    //分配zend_execute_data
    execute_data = zend_vm_stack_push_call_frame(ZEND_CALL_TOP_C
ODE,
            (zend_function*)op_array, 0, zend_get_called_scope(E
G(current_execute_data)), zend_get_this_object(EG(current_execut
e_data)));
    if (EG(current_execute_data)) {
        execute_data->symbol_table = zend_rebuild_symbol_table()
;
    } else {
        execute_data->symbol_table = &EG(symbol_table);
    }
    EX(prev_execute_data) = EG(current_execute_data); //=> execu
te_data->prev_execute_data = EG(current_execute_data);
    i_init_execute_data(execute_data, op_array, return_value); /
/初始化execute_data
    zend_execute_ex(execute_data); //执行opcode
    zend_vm_stack_free_call_frame(execute_data); //释放execute_da
ta:销毁所有的PHP变量
}
```

上面的过程分为四步：

(1)分配stack

由 `zend_vm_stack_push_call_frame` 函数分配一块用于当前作用域的内存空间，返回结果是 `zend_execute_data` 的起始位置。

```
//zend_execute.h
static zend_always_inline zend_execute_data *zend_vm_stack_push_call_frame(uint32_t call_info, zend_function *func, uint32_t num_args, ...)
{
    uint32_t used_stack = zend_vm_calc_used_stack(num_args, func);

    return zend_vm_stack_push_call_frame_ex(used_stack, call_info,
        func, num_args, called_scope, object);
}
```

首先根据`zend_execute_data`、当前`zend_op_array`中局部/临时变量数计算需要的内存空间：

```
//zend_execute.h
static zend_always_inline uint32_t zend_vm_calc_used_stack(uint32_t num_args, zend_function *func)
{
    uint32_t used_stack = ZEND_CALL_FRAME_SLOT + num_args; //内部函数只用这么多，临时变量是编译过程中根据PHP的代码优化出的值，比如:`"hi~".time()`，而在内部函数中则没有这种情况

    if (EXPECTED(ZEND_USER_CODE(func->type))) { //在php脚本中写的function
        used_stack += func->op_array.last_var + func->op_array.T - MIN(func->op_array.num_args, num_args);
    }
    return used_stack * sizeof(zval);
}

//zend_compile.h
#define ZEND_CALL_FRAME_SLOT \
    ((int)((ZEND_MM_ALIGNED_SIZE(sizeof(zend_execute_data)) + ZEND_MM_ALIGNED_SIZE(sizeof(zval)) - 1) / ZEND_MM_ALIGNED_SIZE(sizeof(zval))))
```

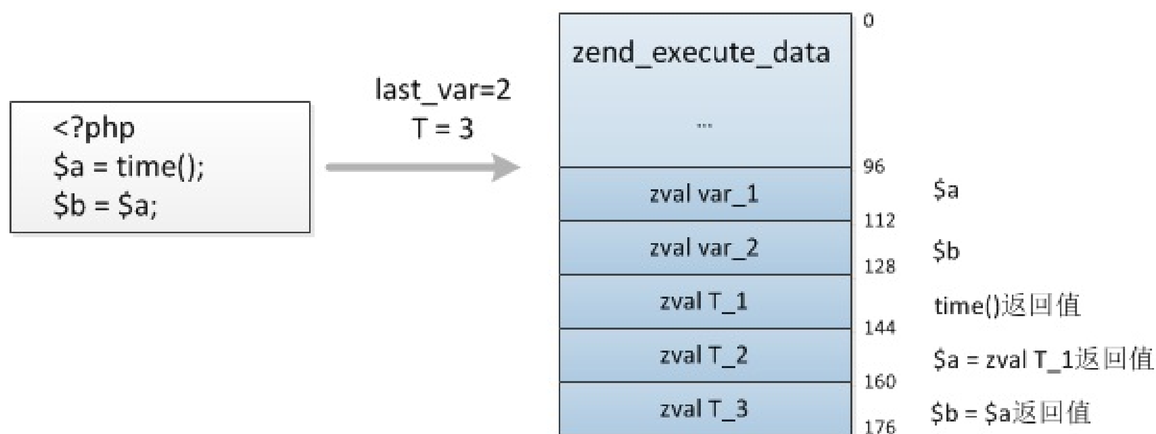
回想下前面编译阶段zend_op_array的结果，在编译过程中已经确定当前作用域下有多少个局部变量(func->op_array.last_var)、临时/中间/无用变量(func->op_array.T)，从而在执行之初就将他们全部分配完成：

- **last_var**：PHP代码中定义的变量数，zend_op.op{1|2}_type = IS_CV 或 result_type & IS_CV的全部数量
- **T**：表示用到的临时变量、无用变量等，zend_op.op{1|2}_type = IS_TMP_VAR|IS_VAR 或resulte_type & (IS_TMP_VAR|IS_VAR)的全部数量

比如赋值操作：`$a = 1234;`，编译后`last_var = 1，T = 1`，`last_var`有`$a`，这里为什么会有`T`？因为赋值语句有一个结果返回值，只是这个值没有用到，假如这么用结果就会用到了`if(($a = 1234) == true){...}`，这时候`$a = 1234;`的返回结果类型是`IS_VAR`，记在`T`上。

`num_args`为函数调用时的实际传入参数数量，`func->op_array.num_args`为全部参数数量，所以`MIN(func->op_array.num_args, num_args)`等于`num_args`，在自定义函数中`used_stack=ZEND_CALL_FRAME_SLOT + func->op_array.last_var + func->op_array.T`，而在调用内部函数时则只需要分配实际传入参数的空间即可，内部函数不会有临时变量的概念。

最终分配的内存空间如下图：



这里实际分配内存时并不是直接`malloc`的，还记得上面EG结构中有个`vm_stack`吗？实际内存是从这里获取的，每次从`EG(vm_stack_top)`处开始分配，分配完再将此指针指向`EG(vm_stack_top) + used_stack`，这里不再对vm_stack作更多分析，更下层实际就是Zend的内存池(zend_alloc.c)，后面也会单独分析。

```
static zend_always_inline zend_execute_data *zend_vm_stack_push_
call_frame_ex(uint32_t used_stack, ...)
{
    zend_execute_data *call = (zend_execute_data*)EG(vm_stack_to
p);
    ...

    //当前vm_stack是否够用
    if (UNEXPECTED(used_stack > (size_t)(((char*)EG(vm_stack_end
)) - (char*)call))) {
        call = (zend_execute_data*)zend_vm_stack_extend(used_sta
ck); //新开辟一块vm_stack
        ...
    }else{ //空间够用，直接分配
        EG(vm_stack_top) = (zval*)((char*)call + used_stack);
        ...
    }

    call->func = func;
    ...
    return call;
}
```

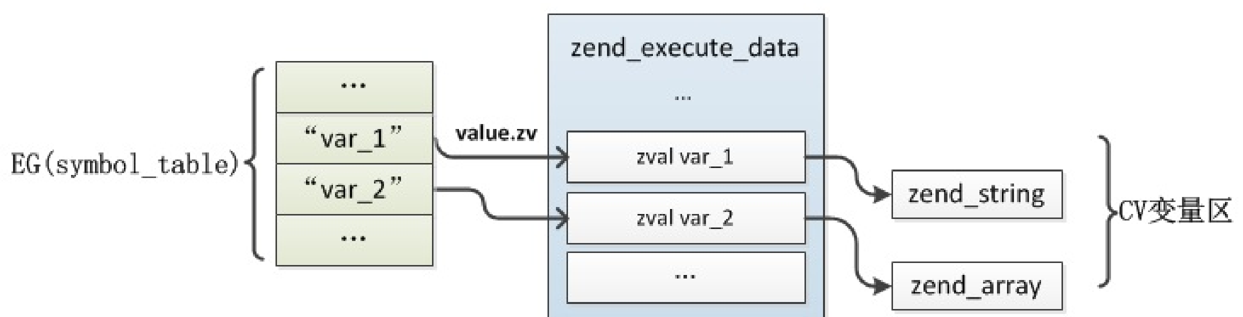
(2)初始化zend_execute_data

注意，这里的初始化是整个php脚本最初的那个，并不是指函数调用时的，这一步的操作主要是设置几个指针: `opline` 、 `call` 、 `return_value` ，同时将PHP的全局变量添加到 `EG(symbol_table)` 中去：

```
//zend_execute.c
static zend_always_inline void i_init_execute_data(zend_execute_data *execute_data, zend_op_array *op_array, zval *return_value)
{
    EX(opline) = op_array->opcodes;
    EX(call) = NULL;
    EX(return_value) = return_value;

    if (UNEXPECTED(EX(symbol_table) != NULL)) {
        ...
        zend_attach_symbol_table(execute_data);//将全局变量添加到EG(symbol_table)中一份，因为此处的execute_data是PHP脚本最初的那个，不是function的，所以所有的变量都是全局的
    }else{ //这个分支的情况还未深入分析，后面碰到再补充
        ...
    }
}
```

`zend_attach_symbol_table()` 的作用是把当前作用域下的变量添加到EG(symbol_table)哈希表中，也就是全局变量，函数中通过global关键词获取的全局变量正是在此时添加的，EG(symbol_table)中的值间接的指向 `zend_execute_data` 中的局部变量，两者的结构如下图所示：



(3)执行opcode

这一步开始具体执行opcode指令，这里调用的是 `zend_execute_ex` ，这是一个函数指针，如果此指针没有被任何扩展重新定义那么将由默认的 `execute_ex` 处理：

```
# define ZEND_OPCODE_HANDLER_ARGS_PASSTHRU execute_data

ZEND_API void execute_ex(zend_execute_data *ex)
{
    zend_execute_data *execute_data = ex;

    while(1) {
        int ret;
        if (UNEXPECTED((ret = ((opcode_handler_t)EX(opline)->han
dler)(execute_data /*ZEND_OPCODE_HANDLER_ARGS_PASSTHRU*/)) != 0)
) {
            if (EXPECTED(ret > 0)) { //调到新的位置执行：函数调用时的
情况
                execute_data = EG(current_execute_data);
            }else{
                return;
            }
        }
    }
}
```

大概的执行过程上面已经介绍过了，这里只分析下整体执行流程，至于PHP各语法具体的handler处理后面会单独列一章详细分析。

(4)释放stack

这一步就比较简单了，只是将申请的 `zend_execute_data` 内存释放给内存池(注意这里并不是变量的销毁)，具体的操作只需要修改几个指针即可：

```
static zend_always_inline void zend_vm_stack_free_call_frame_ex(
uint32_t call_info, zend_execute_data *call)
{
    ZEND_ASSERT_VM_STACK_GLOBAL;

    if (UNEXPECTED(call_info & ZEND_CALL_ALLOCATED)) {
        zend_vm_stack p = EG(vm_stack);

        zend_vm_stack prev = p->prev;

        EG(vm_stack_top) = prev->top;
        EG(vm_stack_end) = prev->end;
        EG(vm_stack) = prev;
        efree(p);

    } else {
        EG(vm_stack_top) = (zval*)call;
    }

    ZEND_ASSERT_VM_STACK_GLOBAL;
}

static zend_always_inline void zend_vm_stack_free_call_frame(zen
d_execute_data *call)
{
    zend_vm_stack_free_call_frame_ex(ZEND_CALL_INFO(call), call)
;
}
```

3.3.3 函数的执行流程

（这里的函数指用户自定义的PHP函数，不含内部函数） 上一节我们介绍了zend执行引擎的几个关键步骤，也简单的介绍了函数的调用过程，这里再单独总结下：

- 【初始化阶段】 这个阶段首先查找到函数的zend*function*，普通*function*就是到*EG(functiontable)*中查找，成员方法则先从*EG(class_table)*中找到*zend_class_entry*，然后再进一步在其*function_table*找到*zend_function*，接着就是根据*zend_op_array*新分配 *__zend_execute_data* 结构并设置上下文切换

的指针
- 【参数传递阶段】 如果函数没有参数则跳过此步骤，有的话则会将函数所需参数传递到 初始化阶段 新分配的 **zend_execute_data**动态变量区
- 【函数调用阶段】 这个步骤主要是做上下文切换，将执行器切换到调用的函数上，可以理解会在这个阶段递归调用**zend_execute_ex**函数实现call的过程(实际并一定是递归，默认是在while(1){...}中切换执行空间的，但如果我们在扩展中重定义了zend_execute_ex用来介入执行流程则就是递归调用)
- 【函数执行阶段】 被调用函数内部的执行过程，首先是接收参数，然后开始执行opcode
- 【函数返回阶段】 被调用函数执行完毕返回过程，将返回值传递给调用方的zend_execute_data变量区，然后释放zend_execute_data以及分配的局部变量，将上下文切换到调用前，回到调用的位置继续执行，这个实际是函数执行中的一部分，不算是独立的一个过程

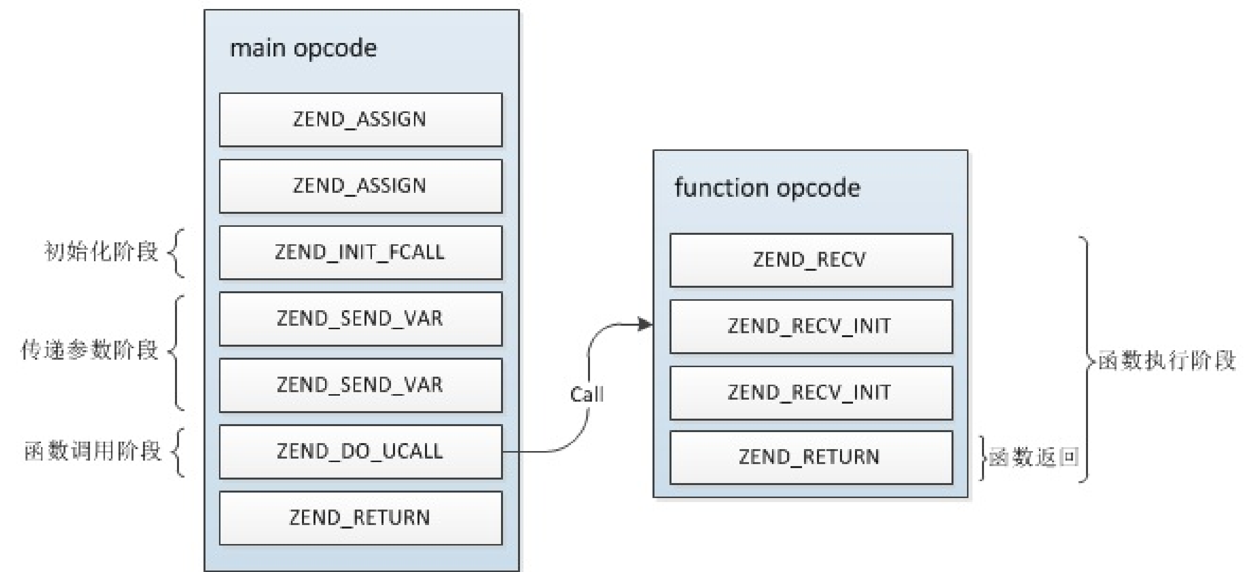
接下来我们一个具体的例子详细分析下各个阶段的处理过程：

```
function my_function($a, $b = false, $c = "hi"){
    return $c;
}

$a = array();
$b = true;

my_function($a, $b);
```

主脚本、my_function的opcode为：



3.3.3.1 初始化阶段

此阶段的主要工作有两个：查找函数zend_function、分配zend_execute_data。

上面的例子此过程执行的opcode为 `ZEND_INIT_FCALL` ，根据op_type计算可得handler为 `ZEND_INIT_FCALL_SPEC_CONST_HANDLER` ：

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_INIT_FCALL_SPE
C_CONST_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    USE_OPLINE

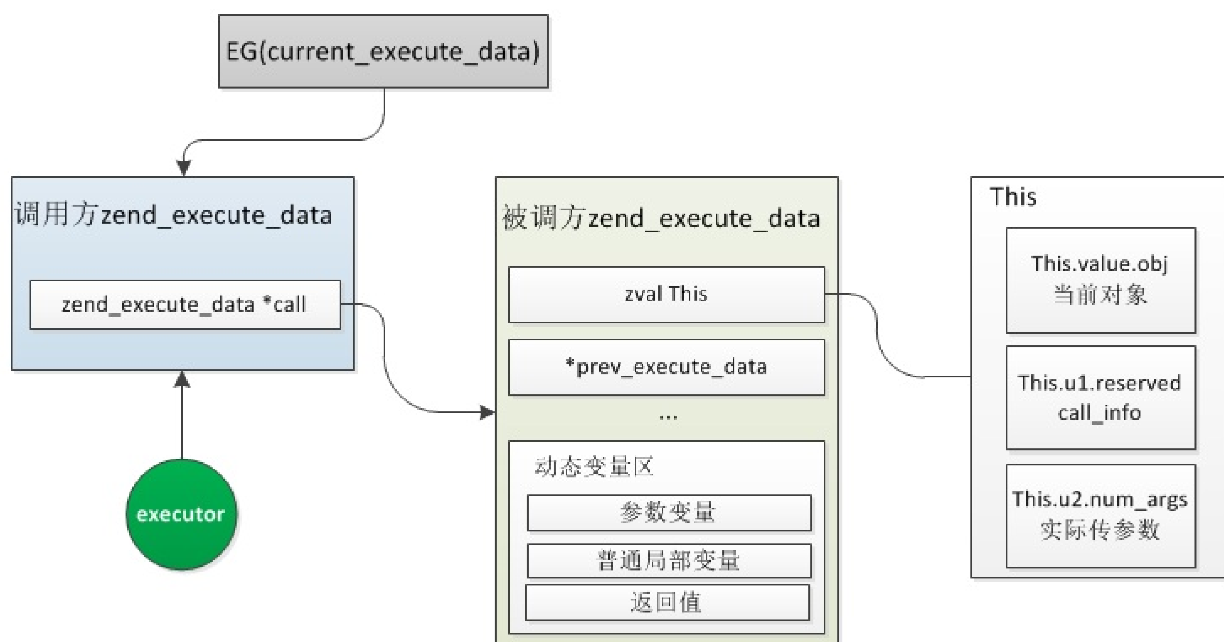
    zval *fname = EX_CONSTANT(opline->op2); //调用的函数名称通过操作
数2记录
    zval *func;
    zend_function *fbc;
    zend_execute_data *call;

    //这里牵扯到zend的一种缓存机制：运行时缓存，后面我们会单独分析，这里忽
略即可
    ...
        //首先根据函数名去EG(function_table)索引zend_function
        func = zend_hash_find(EG(function_table), Z_STR_P(fname)
);
        if (UNEXPECTED(func == NULL)) {
            SAVE_OPLINE();
            zend_throw_error(NULL, "Call to undefined function %
s()", Z_STRVAL_P(fname));
            HANDLE_EXCEPTION();
        }
        fbc = Z_FUNC_P(func); //(*func).value.func
    ...

    //分配zend_execute_data
    call = zend_vm_stack_push_call_frame_ex(
        opline->op1.num, ZEND_CALL_NESTED_FUNCTION,
        fbc, opline->extended_value, NULL, NULL);
    call->prev_execute_data = EX(call);
    EX(call) = call; //将当前正在运行的zend_execute_data.call指向新
分配的zend_execute_data

    ZEND_VM_NEXT_OPCODE();
}
```

当前zend_execute_data及新生成的zend_execute_data关系：



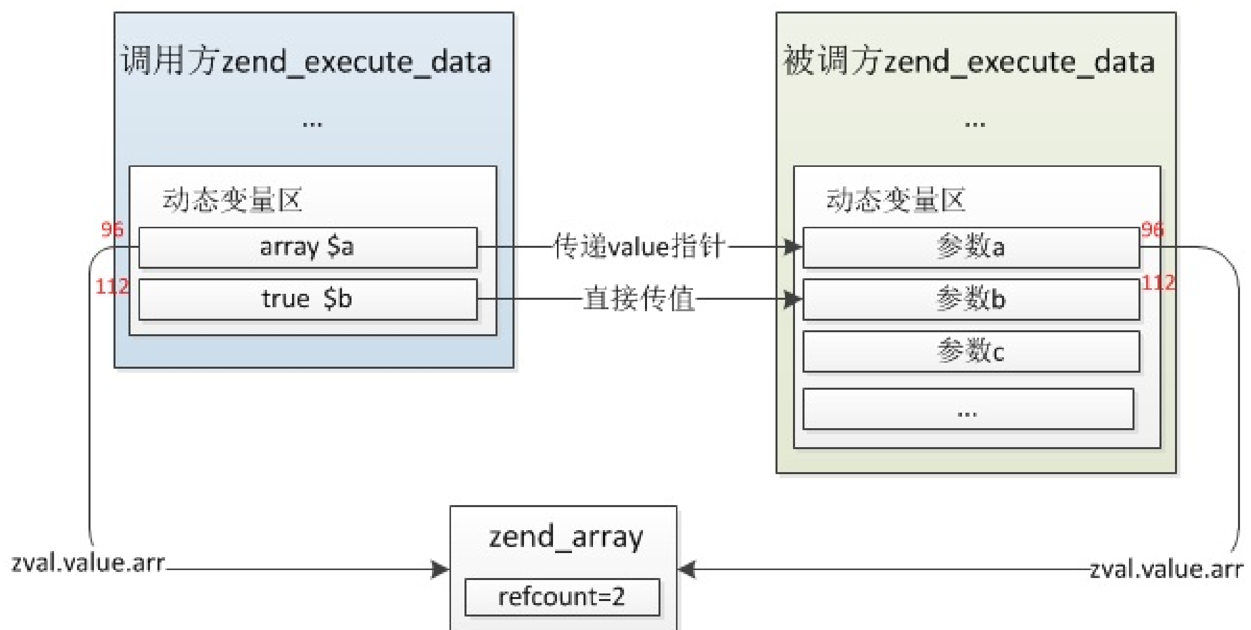
注意 **This** 这个值，它并不仅仅指的是面向对象中那个this，此外它还记录着其它两个信息：

- **call_info**： 调用信息，通过 **This.u1.reserved** 记录，因为我们的主脚本、用户自定义函数调用、内核函数调用、include/require/eval等都会生成一个zend_execute_data，这个值就是用来区分这些不同类型的，对应的具体值为：ZEND_CALL_TOP_CODE、ZEND_CALL_NESTED_FUNCTION、ZEND_CALL_TOP_FUNCTION、ZEND_CALL_NESTED_CODE，这个信息是在分配zend_execute_data时显式声明的
- **num_args**： 函数调用实际传入的参数数量，通过 **This.u2.num_args** 记录，比如示例中我们定义的函数有3个参数，其中1个是必传的，而我们调用时传入了2个，所以这个例子中的num*args*就是*2*，这个值在编译时知道的，保存在 *_zend_op->extended_value* 中

3.3.3.2 参数传递阶段

这个过程就是将当前作用空间下的变量值"复制"到新的zend_execute_data动态变量区中，那么调用方怎么知道要把值传递到新zend_execute_data哪个位置呢？实际这个地方是有固定规则的，zend_execute_data的动态变量区最前面是参数变量，按照参数的顺序依次分配，接着才是普通的局部变量、临时变量等，所以调用方就可以根据传的是第几个参数来确定其具体的存储位置。

另外这里的"复制"并不是硬拷贝，而是传递的value指针(当然bool/int/double类型不需要)，通过引用计数管理，当在被调函数内部改写参数的值时将重新拷贝一份，与普通的变量用法相同。



图中画的只是上面示例那种情况，比如 `my_function(array());` 直接传值则会是**literals**区**->**新**zend_execute_data**动态变量区 的传递。

3.3.3.3 函数调用阶段

这个过程主要是进行一些上下文切换，将执行器切换到调用的函数上。

上面例子对应的opcode为 `ZEND_DO_UCALL` ，handler为 `ZEND_DO_UCALL_SPEC_HANDLER` ：

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_DO_UCALL_SPEC_
HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    USE_OPLINE
    zend_execute_data *call = EX(call);
    zend_function *fbc = call->func;
    zval *ret;

    SAVE_OPLINE();
    EX(call) = call->prev_execute_data;

    EG(scope) = NULL;
```

```
    ret = NULL;
    call->symbol_table = NULL;
    if (RETURN_VALUE_USED(opline)) {
        ret = EX_VAR(opline->result.var); //函数返回值的存储位置
        ZVAL_NULL(ret);
        Z_VAR_FLAGS_P(ret) = 0;
    }

    call->prev_execute_data = execute_data; //将新zend_execute_da
ta->prev_execute_data指向当前data
    i_init_func_execute_data(call, &fbc->op_array, ret, 0);

    ZEND_VM_ENTER();
}

//zend_execute.c
static zend_always_inline void i_init_func_execute_data(zend_exe
cute_data *execute_data, zend_op_array *op_array, zval *return_v
alue, int check_this)
{
    uint32_t first_extra_arg, num_args;
    ZEND_ASSERT(EX(func) == (zend_function*)op_array);

    EX(opline) = op_array->opcodes;
    EX(call) = NULL;
    EX(return_value) = return_value;

    first_extra_arg = op_array->num_args; //函数的总参数数量，示例中
为3
    num_args = EX_NUM_ARGS(); //实际传入参数数量，示例中为2
    if (UNEXPECTED(num_args > first_extra_arg)) {
        ...
    } else if (EXPECTED((op_array->fn_flags & ZEND_ACC_HAS_TYPE_
HINTS) == 0)) {
        //跳过前面几个已经传参的参数接收的指令，因为已经显式的传递参数了，
无需再接收默认值
        EX(opline) += num_args;
    }

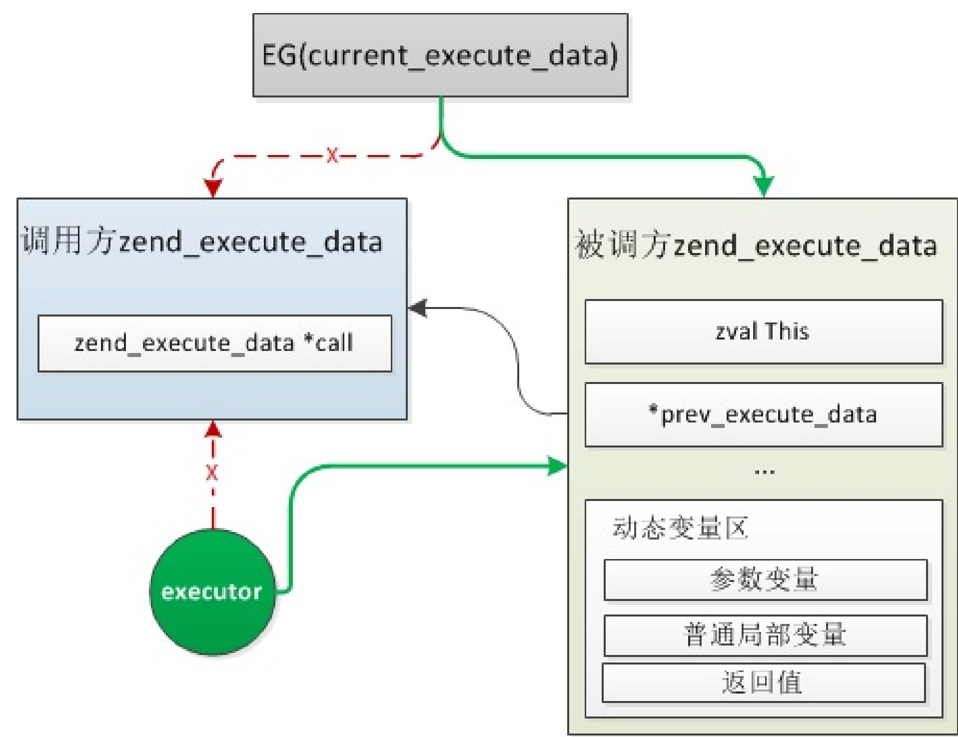
    //初始化动态变量区，将所有变量(除已经传入的外)设置为IS_UNDEF
```

```
    if (EXPECTED((int)num_args < op_array->last_var)) {
        zval *var = EX_VAR_NUM(num_args);
        zval *end = EX_VAR_NUM(op_array->last_var);

        do {
            ZVAL_UNDEF(var);
            var++;
        } while (var != end);
    }
    ...

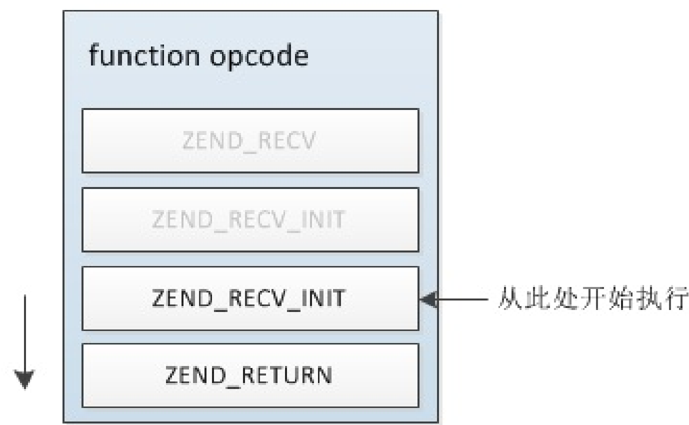
    //分配运行时缓存，此机制后面再单独说明
    if (UNEXPECTED(!op_array->run_time_cache)) {
        op_array->run_time_cache = zend_arena_alloc(&CG(arena),
op_array->cache_size);
        memset(op_array->run_time_cache, 0, op_array->cache_size
);
    }
    EX_LOAD_RUN_TIME_CACHE(op_array); //execute_data.run_time_ca
che = op_array.run_time_cache
    EX_LOAD_LITERALS(op_array); //execute_data.literals = op_arr
ay.literals

    //EG(current_execute_data)为执行器当前执行空间，将执行器切到函数内
    EG(current_execute_data) = execute_data;
}
```



3.3.3.4 函数执行阶段

这个过程就是函数内部opcode的执行流程，没什么特别的，唯一的不同就是前面会接收未传的参数，如下图所示。



3.3.3.5 函数返回阶段

实际此过程可以认为是3.3.3.4的一部分，这个阶段就是函数调用结束，返回调用处的过程，这个过程中有三个关键工作：拷贝返回值、执行器切回调用位置、释放清理局部变量。

上面例子此过程opcode为 `ZEND_RETURN` ，对应的handler为 `ZEND_RETURN_SPEC_CV_HANDLER` ：

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_RETURN_SPEC_CV_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    USE_OPLINE
    zval *retval_ptr;
    zend_free_op free_op1;

    //获取返回值
    retval_ptr = _get_zval_ptr_cv_undef(execute_data, opline->op1.var);
    if (IS_CV == IS_CV && UNEXPECTED(Z_TYPE_INFO_P(retval_ptr) == IS_UNDEF)) {
        //返回值未定义，返回NULL
        retval_ptr = GET_OP1_UNDEF_CV(retval_ptr, BP_VAR_R);
        if (EX(return_value)) {
            ZVAL_NULL(EX(return_value));
        }
    } else if(!EX(return_value)){
        //无返回值
        ...
    }else{ //返回值正常
        ...

        ZVAL_DEREF(retval_ptr); //如果retval_ptr是引用则将找到其具体引用的zval
        ZVAL_COPY(EX(return_value), retval_ptr); //将返回值复制给调用方接收值：EX(return_value)
        ...
    }

    ZEND_VM_TAIL_CALL(zend_leave_helper_SPEC(ZEND_OPCODE_HANDLER_ARGS_PASSTHRU));
}
```

继续看下`zend_leave_helper_SPEC`，执行器切换、局部变量清理就是在这个函数中完成的。

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL zend_leave_helper_SPEC(ZEND_OPCODE_HANDLER_ARGS)
```

```
{
    zend_execute_data *old_execute_data;
    uint32_t call_info = EX_CALL_INFO();

    if (EXPECTED(ZEND_CALL_KIND_EX(call_info) == ZEND_CALL_NESTE
D_FUNCTION)) {
        //普通的函数调用将走到这个分支

        i_free_compiled_variables(execute_data);
        ...
    }
    //include、eval及整个脚本的结束(main函数)走到下面
    //...

    //将执行器切回调用的位置
    EG(current_execute_data) = EX(prev_execute_data);
}

//zend_execute.c
//清理局部变量的过程
static zend_always_inline void i_free_compiled_variables(zend_ex
ecute_data *execute_data)
{
    zval *cv = EX_VAR_NUM(0);
    zval *end = cv + EX(func)->op_array.last_var;
    while (EXPECTED(cv != end)) {
        if (Z_REFCOUNTED_P(cv)) {
            if (!Z_DELREF_P(cv)) { //引用计数减一后为0
                zend_refcounted *r = Z_COUNTED_P(cv);
                ZVAL_NULL(cv);
                zval_dtor_func_for_ptr(r); //释放变量值
            } else {
                GC_ZVAL_CHECK_POSSIBLE_ROOT(cv); //引用计数减一后>
0，启动垃圾检查机制，清理循环引用导致无法回收的垃圾
            }
        }
        cv++;
    }
}
```

除了函数调用完成时有return操作，其它还有两种情况也会有此过程：

- **1.PHP**主脚本执行结束时：也就是PHP脚本开始执行的入口脚本(PHP没有显式的main函数，这种就可以认为是main函数)，但是这种情况并不会在return时清理，因为在main函数中定义的变量并非纯碎的局面变量，它们都是全局变量，与$**GET**、**$**POST是一类，这些全局变量的清理是在request_shutdown阶段处理
- **2.include**、**eval**：以include为例，如果include的文件中定义了全局变量，那么这些变量实际与上面1的情况一样，它们的存储位置是在一起的

所以实际上面说的这两种情况属于一类，它们并不是局部变量的清理，而是 全局变量的清理 ，另外局部变量的清理也并非只有return一个时机，另外还有一个更重要的时机就是变量分离时，这种情况我们在《PHP语法实现》一节再具体说明。

3.3.4 全局execute_data和opline

Zend执行器在opcode的执行过程中，会频繁的用到execute_data和opline两个变量，execute_data为zend_execute_data结构，opline为当前执行的指令。普通的处理方式在执行每条opcode指令的handler时，会把execute_data地址作为参数传给handler使用，使用时先从当前栈上获取execute_data地址，然后再从堆上获取变量的数据，这种方式下Zend执行器展开后是下面这样：

```
ZEND_API void execute_ex(zend_execute_data *ex)
{
    zend_execute_data *execute_data = ex;

    while (1) {
        int ret;

        if (UNEXPECTED((ret = ((opcode_handler_t)execute_data->o
pline->handler)(execute_data)) != 0)) {
            if (EXPECTED(ret > 0)) {
                execute_data = EG(current_execute_data);
            } else {
                return;
            }
        }
    }
}
```

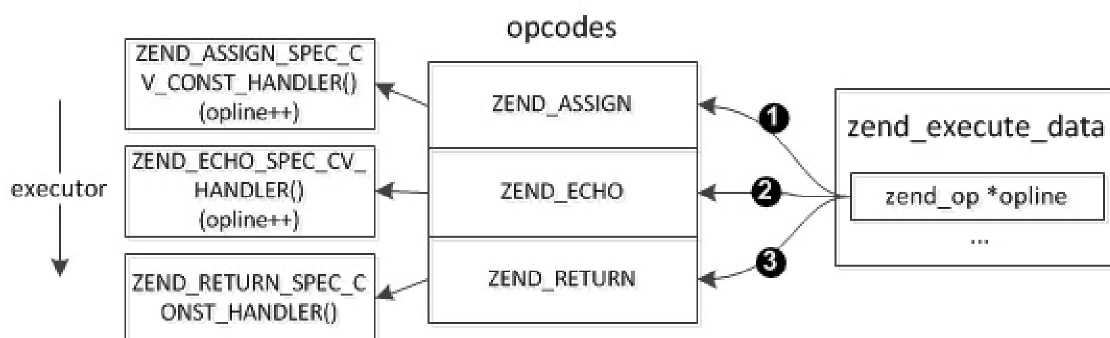
执行器实际是一个大循环，从第一条opcode开始执行，execute_data->opline指向当前执行的指令，执行完以后指向下一条指令，opline类似eip(或rip)寄存器的作用。通过这个循环，ZendVM完成opcode指令的执行。opcode执行完后以后指向下一条指令的操作是在当前handler中完成，也就是说每条执行执行完以后会主动更新opline，这里会有下面几个不同的动作：

```
#define ZEND_VM_CONTINUE()     return  0
#define ZEND_VM_ENTER()        return  1
#define ZEND_VM_LEAVE()        return  2
#define ZEND_VM_RETURN()       return -1
```

ZEND_VM_CONTINUE()表示继续执行下一条opcode；ZEND_VM_ENTER()/ZEND_VM_LEAVE()是调用函数时的动作，普通模式下ZEND_VM_ENTER()实际就是return 1，然后execute_ex()中会将execute_data切换到被调函数的结构上，对应的，在函数调用完成后ZEND_VM_LEAVE()会return 2，再将execute_data切换至原来的结构；ZEND_VM_RETURN()表示执行完成，返回-1给execute_ex()，比如exit，这时候execute_ex()将退出执行。下面看一个具体的例子：

```
$a = "hi~";
echo $a;
```

执行过程如下图所示：



以ZEND_ASSIGN这条赋值指令为例，其handler展开前如下：

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_ASSIGN_SPEC_CV
_CONST_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    USE_OPLINE
    ...
    ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION();
}
```

所有opcode的handler定义格式都是相同的，其参数列表通过ZEND_OPCODE_HANDLER_ARGS宏定义，展开后实际只有一个execute_data，展开后：

```
static int ZEND_ASSIGN_SPEC_CV_CONST_HANDLER(zend_execute_data *
execute_data)
{
    //USE_OPLINE
    const zend_op *opline = execute_data->opline;
    ...

    //ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION()
    execute_data->opline = execute_data->opline + 1;
    return 0;
}
```

从这个例子可以很清楚的看到，执行完以后会将execute_data->opline加1，也就是指向下一条opcode，然后返回0给execute_ex()，接着执行器在下一次循环时执行下一条opcode，依次类推，直至所有的opcode执行完成。这个处理过程比较简单，并没有不好理解的地方，而且整个过程看起来也都那么顺理成章。PHP7针对execute_data、opline两个变量的存储位置进行了优化，那就是使用全局寄存器保存这两个变量的地址，以实现更高效率的读取。这种方式下execute_data、opline直接从寄存器读取地址，在性能上大概有5%的提升(官方说法)。在分析PHP7的优化之前，我们先简单介绍下什么是寄存器变量。

寄存器变量存放在CPU的寄存器中，使用时，不需要访问内存直接从寄存器中读写，与存储在内存中的变量相比，寄存器变量具有更快的访问速度，在计算机的存储层次中，寄存器的速度最快，其次是内存，最慢的是硬盘。C语言中使用关键字register来声明局部变量为寄存器变量，需要注意的是，只有局部自动变量和形式参数才能够被定义为寄存器变量，全局变量和局部静态变量都不能被定义为寄存器变量。而且，一个计算机中寄存器数量是有限的，一般为2到3个，因此寄存器变量的数量不能太多。对于在一个函数中说明的多于2到3个的寄存器变量，C编译程序会自动地将寄存器变量变为自动变量。 受硬件寄存器长度的限制，寄存器变量只能是char、int或指针型，而不能使其他复杂数据类型。由于register变量使用的是硬件CPU中的寄存器，寄存器变量无地址，所以不能使用取地址运算符"&"求寄存器变量的地址。

GCC从4.8.0版本开始支持了另外一项特性：全局寄存器变量(Global Register Variables，详细介绍)，也就是可以把全局变量定义为寄存器变量，从而可以实现函数间共享数据。可以通过下面的语法告诉编译器使用寄存器来保存数据：

```
register int *foo asm ("r12"); //r12、%r12
```

或者：

```
register int *foo __asm__ ("r12"); //r12、%r12
```

这里r12就是指定使用的寄存器，它必须是运行平台上有效的寄存器，这样就可以像使用普通的变量一样使用foo，但是foo同样没有地址，也就是无法通过&获取它的地址，在gdb调试时也无法使用foo符号，只能使用对应的寄存器获取数据。举个例子来看：

```
//main.c
#include <stdlib.h>

typedef struct _execute_data {
    int ip;
}zend_execute_data;


register zend_execute_data* execute_data __asm__ ("%r14");

int main(void)
{
    execute_data = (zend_execute_data  *)malloc(sizeof(zend_exec
ute_data));
    execute_data->ip = 9999;

    return 0;
}
```

编译：`$ gcc -o main -g main.c`，然后通过gdb看下：

```
$ gdb main
(gdb) break main
(gdb) r
Starting program: /home/qinpeng/c/php/main

Breakpoint 1, main () at main.c:12
12	execute_data = (zend_execute_data  *)malloc(sizeof(zend_
execute_data));
(gdb) n
13	execute_data->ip = 9999;
(gdb) n
15	return 0;
```

这时我们就无法再像普通变量那样直接使用execute_data访问数据，只能通过r14寄存器读取：

```
(gdb) p execute_data
Missing ELF symbol "execute_data".
(gdb) info register r14
r14            0x601010 6295568
(gdb) p ((zend_execute_data *)$r14)->ip
$3 = 9999
```

了解完全局寄存器变量，接下来我们再回头看下PHP7中的用法，处理也比较简单，就是在execute_ex()执行各opcode指令的过程中，不再将execute_data作为参数传给handler，而是通过寄存器保存execute_data及opline的地址，handler使用时直接从全局变量(寄存器)读取，执行完再把下一条指令更新到全局变量。

该功能需要GCC 4.8+支持，默认开启，可以通过 --disable-gcc-global-regs 编译参数关闭。以x86_64为例，execute_data使用r14寄存器，opline使用r15寄存器：

```
//file: zend_execute.c line: 2631
# define ZEND_VM_FP_GLOBAL_REG "%r14"
# define ZEND_VM_IP_GLOBAL_REG "%r15"

//file: zend_vm_execute.h line: 315
register zend_execute_data* volatile execute_data __asm__(ZEND_V
M_FP_GLOBAL_REG);
register const zend_op* volatile opline __asm__(ZEND_VM_IP_GLOBA
L_REG);
```

execute_data、opline定义为全局变量，下面看下execute_ex()的变化，展开后：

```
ZEND_API void execute_ex(zend_execute_data *ex)
{
    const zend_op *orig_opline = opline;
    zend_execute_data *orig_execute_data = execute_data;

    //将当前execute_data、opline保存到全局变量
    execute_data = ex;
    opline = execute_data->opline

    while (1) {
        ((opcode_handler_t)opline->handler)();

        if (UNEXPECTED(!opline)) {
            execute_data = orig_execute_data;
            opline = orig_opline;

            return;
        }
    }
}
```

这个时候调用各opcode指令的handler时就不再传入execute_data的参数了，handler使用时直接从全局变量读取，仍以上面的赋值ZEND_ASSIGN指令为例，handler展开后：

```
static int ZEND_ASSIGN_SPEC_CV_CONST_HANDLER(void)
{
    ...

    //ZEND_VM_NEXT_OPCODE_CHECK_EXCEPTION()
    opline = execute_data->opline + 1;
    return;
}
```

当调用函数时，会把execute_data、opline更新为被调函数的，然后回到execute_ex()开始执行被调函数的指令：

```
# define ZEND_VM_ENTER()           execute_data = EG(current_exe
cute_data); LOAD_OPLINE(); ZEND_VM_CONTINUE()
```

展开后：

```
//ZEND_VM_ENTER()
execute_data = execute_data->current_execute_data;
opline = execute_data->opline;
return;
```

这两种处理方式并没有本质上的差异，只是通过全局寄存器变量提升了一些性能。

> **Note:** automake编译时的命令是cc，而不是gcc，如果更新gcc后发现PHP仍然没有支持这个特性，请检查下cc是否指向了新的gcc

3.4.1 类

类是现实世界或思维世界中的实体在计算机中的反映，它将某些具有关联关系的数据以及这些数据上的操作封装在一起。在面向对象中类是对象的抽象，对象是类的具体实例。

在PHP中类编译阶段的产物，而对象是运行时产生的，它们归属于不同阶段。

PHP中我们这样定义一个类：

```
class 类名 {
    常量;
    成员属性;
    成员方法;
}
```

一个类可以包含有属于自己的常量、变量（称为“属性”）以及函数（称为“方法”），本节将围绕这三部分具体弄清楚以下几个问题：

- a.类的存储及索引
- b.成员属性的存储结构
- c.成员方法的存储结构
- d.成员方法的调用过程及与普通function调用的差别

3.4.1.1 类的结构及存储

首先我们看下类的数据结构：

```
struct _zend_class_entry {
    char type;          //类的类型：内部类ZEND_INTERNAL_CLASS(1)、用户自定义类ZEND_USER_CLASS(2)
    zend_string *name;  //类名，PHP类不区分大小写，统一为小写
    struct _zend_class_entry *parent; //父类
    int refcount;
    uint32_t ce_flags;  //类掩码，如普通类、抽象类、接口，除了这还有别的含义，暂未弄清

    int default_properties_count;        //普通属性数，包括public、
```

```
private
    int default_static_members_count;    //静态属性数，static
    zval *default_properties_table;      //普通属性值数组
    zval *default_static_members_table;  //静态属性值数组
    zval *static_members_table;
    HashTable function_table;  //成员方法哈希表
    HashTable properties_info; //成员属性基本信息哈希表，key为成员名，
value为zend_property_info
    HashTable constants_table; //常量哈希表，通过constant定义的

    //以下是构造函授、析构函数、魔术方法的指针
    union _zend_function *constructor;
    union _zend_function *destructor;
    union _zend_function *clone;
    union _zend_function *__get;
    union _zend_function *__set;
    union _zend_function *__unset;
    union _zend_function *__isset;
    union _zend_function *__call;
    union _zend_function *__callstatic;
    union _zend_function *__tostring;
    union _zend_function *__debugInfo;
    union _zend_function *serialize_func;
    union _zend_function *unserialize_func;

    zend_class_iterator_funcs iterator_funcs;

    //自定义的钩子函数，通常是定义内部类时使用，可以灵活的进行一些个性化的操作

    //用户自定义类不会用到，暂时忽略即可
    zend_object* (*create_object)(zend_class_entry *class_type);
    zend_object_iterator *(*get_iterator)(zend_class_entry *ce,
zval *object, int by_ref);
    int (*interface_gets_implemented)(zend_class_entry *iface, z
end_class_entry *class_type); /* a class implements this interfa
ce */
    union _zend_function *(*get_static_method)(zend_class_entry
*ce, zend_string* method);

    /* serializer callbacks */
```

```
    int (*serialize)(zval *object, unsigned char **buffer, size_t
 *buf_len, zend_serialize_data *data);
    int (*unserialize)(zval *object, zend_class_entry *ce, const
unsigned char *buf, size_t buf_len, zend_unserialize_data *data)
;

    uint32_t num_interfaces; //实现的接口数
    uint32_t num_traits;
    zend_class_entry **interfaces; //实现的接口

    zend_class_entry **traits;
    zend_trait_alias **trait_aliases;
    zend_trait_precedence **trait_precedences;

    union {
        struct {
            zend_string *filename;
            uint32_t line_start;
            uint32_t line_end;
            zend_string *doc_comment;
        } user;
        struct {
            const struct _zend_function_entry *builtin_functions
;
            struct _zend_module_entry *module; //所属扩展
        } internal;
    } info;
}
```

create_object为实例化对象的操作，可以通过扩展自定义一个函数来接管实例化对象的操作，没有定义这个函数的话将由默认的 `zend_objects_new()` 处理，自定义时可以参考这个函数的实现：

```
//注意：此操作并没有将属性拷贝到zend_object中：由object_properties_ini
t()完成
ZEND_API zend_object *zend_objects_new(zend_class_entry *ce)
{
    zend_object *object = emalloc(sizeof(zend_object) + zend_obj
ect_properties_size(ce));

    zend_object_std_init(object, ce);
    //设置对象操作的handler
    object->handlers = &std_object_handlers;
    return object;
}
```

举个例子具体看下，定义一个User类，它继承了Human类，User类中有一个常量、一个静态属性、两个普通属性：

```
//父类
class Human {}

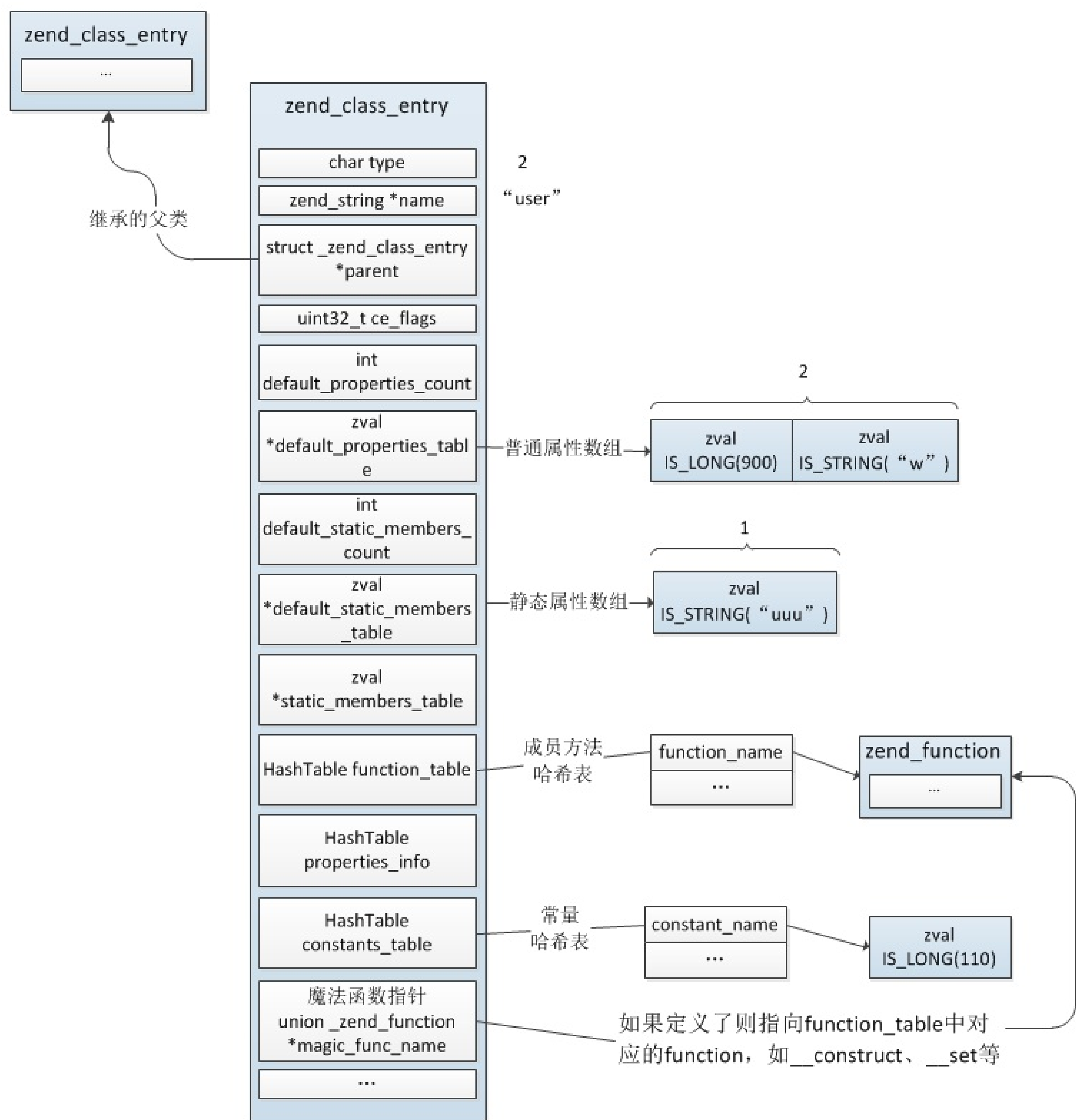
class User extends Human
{
    const type = 110;

    static $name = "uuu";
    public $uid = 900;
    public $sex = 'w';

    public function __construct(){
    }

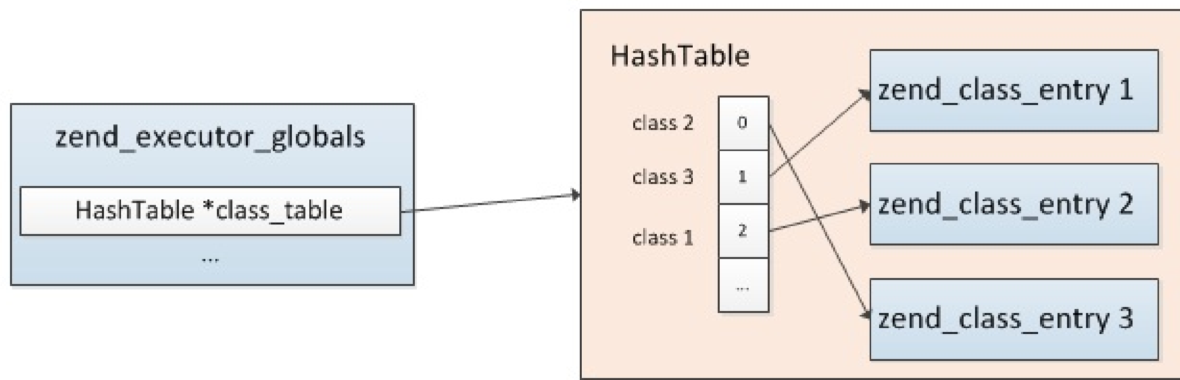
    public function getName(){
        return $this->name;
    }
}
```

其对应的zend_class_entry存储结构如下图。



开始的时候已经提到，类是编译阶段的产物，编译完成后我们定义的每个类都会生成一个zend_class_entry，它保存着类的全部信息，在执行阶段所有类相关的操作都是用的这个结构。

所有PHP脚本中定义的类以及内核、扩展中定义的内部类通过一个以"类名"作为索引的哈希表存储，这个哈希表保存在Zend引擎global变量中：**zend_executor_globals.class_table**(即：**EG(class_table)**)，与function的存储相同，关于这个global变量前面《3.3.1.3 zend_executor_globals》已经讲过。



在接下来的小节中我们将对类的常量、成员属性、成员方法的实现具体分析。

3.4.1.2 类常量

PHP中可以把在类中始终保持不变的值定义为常量，在定义和使用常量的时候不需要使用 $ 符号，常量的值必须是一个定值(如布尔型、整形、字符串、数组，php5.*不支持数组)，不能是变量、数学运算的结果或函数调用，也就是说它是只读的，无法进行赋值。

常量通过 **const** 定义：

```
class my_class {
    const 常量名 = 常量值;
}
```

常量通过 **class_name::**常量名 访问，或在class内部通过 **self::**常量名 访问。

常量是类维度的数据(而不是对象的)，它们通过 `zend_class_entry.constants_table` 进行存储，这是一个哈希结构，通过 常量名 索引，value就是具体定义的常量值。

常量的读取：

根据前面我们对PHP opcode已有的了解，我们可以猜测常量访问的opcode的组成：常量名保存在literals中(其op_type = IS_CONST)，执行时先取出常量名，然后去zend_class_entry.constants_table哈希表中索引到具体的常量值即可。

事实上我们的这个猜测并不是完全正确的，因为有的情况确实是我们猜想的那样，但是还有另外一种情况，比较下两个例子的不同：

```
//示例1
echo my_class::A1;

class my_class {
    const A1 = "hi";
}
```

```
//示例2

class my_class {
    const A1 = "hi";
}

echo my_class::A1;
```

唯一的不同就是常量的使用时机：示例1是在定义前使用的，示例2是在定义后使用的。我们都知道PHP变量无需提前声明，这俩会有什么不同呢？

事实上这两种情况内核会有两种不同的处理方式，示例1这种情况的处理与我们上面的猜测相同，而示例2则有另外一种处理方式：PHP代码的编译是顺序的，示例2的情况编译到 `echo my_class::A1` 这行时首先会尝试检索下是否已经编译了my*class*，如果能在*CG(classtable)*中找到，则进一步从类的 `contants_table` 查找对应的常量，找到的话则会复制其*value*替换常量，简单的讲就是类似*C*语言中的宏，__编译时替换为实际的值了，而不是在运行时再去检索。

具体debug下上面两个例子会发现示例2的主要的opcode只有一个ZEND*ECHO*，也就是直接输出值了，并没有设计类常量的查找，这就是因为编译的时候已经将 ***my_class::A1*** 替换为 _*hi* 了， `echo my_class::A1;` 等同于： `echo "hi";` ；而示例1首先的操作则是ZEND_FETCH_CONSTANT，查找常量，接着才是ZEND_ECHO。

3.4.1.3 成员属性

类的变量成员叫做“属性”。属性声明是由关键字 **public**，**protected** 或者 **private** 开头，然后跟一个普通的变量声明来组成，关于这三个关键字这里不作讨论，后面分析可见性的章节再作说明。

> 【修饰符(public/private/protected/static)】【成员属性名】=【属性默认值】;

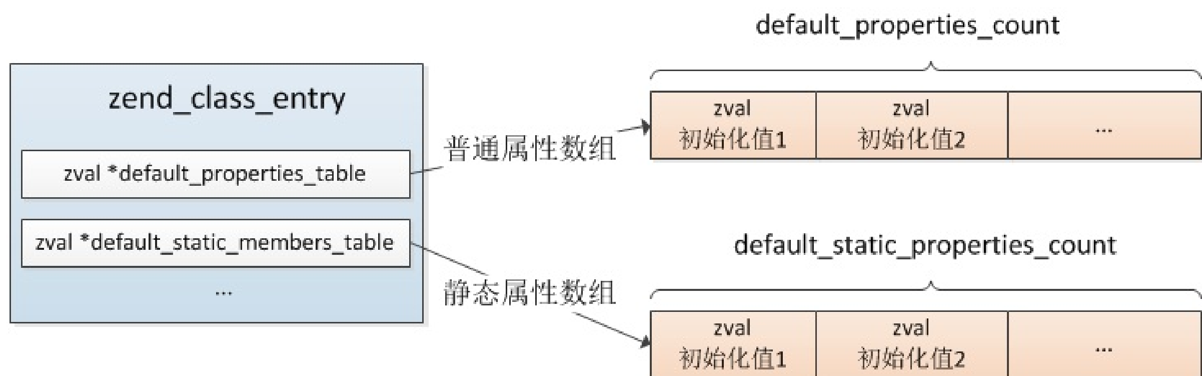
属性中的变量可以初始化，但是初始化的值必须是常数，这里的常数是指 PHP 脚本在编译阶段时就可以得到其值，而不依赖于运行时的信息才能求值，比如 `public $time = time();` 这样定义一个属性就会触发语法错误。

成员属性又分为两类：普通属性、静态属性。静态属性通过 **static** 声明，通过 **self::$property** 或 类名**::$property** 访问；普通属性通过 **$this->property** 或 **$object->property** 访问。

```
class my_class {
    //普通属性
    public $property = 初始化值;

    //静态属性
    public static $property_2 = 初始化值;
}
```

与常量的存储方式不同，成员属性的 初始化值 并不是 直接 用以"属性名"作为索引的哈希表存储的，而是通过数组保存的，普通属性、静态属性各有一个数组分别存储。



看到这里可能有个疑问：使用时成员属性是如果找到的呢？

实际只是成员属性的 **VALUE** 通过数组存储的，访问时仍然是根据以"属性名"为索引的散列表查找具体VALUE的，这个散列表并没有按照普通属性、静态属性分为两个，而是只用了一个：**HashTable properties_info** 。此哈希表存储元素的value类型为 **zend_property_info** 。

```
typedef struct _zend_property_info {
    uint32_t offset; //普通成员变量的内存偏移值
                     //静态成员变量的数组索引
    uint32_t flags;  //属性掩码，如public、private、protected及是否为静态属性
    zend_string *name; //属性名:并不是原始属性名
    zend_string *doc_comment;
    zend_class_entry *ce; //所属类
} zend_property_info;

//flags标识位
#define ZEND_ACC_PUBLIC     0x100
#define ZEND_ACC_PROTECTED  0x200
#define ZEND_ACC_PRIVATE    0x400

#define ZEND_ACC_STATIC         0x01
```

- **name**：属性名，特别注意的是这里并不全是原始属性名，private会在原始属性名前加上类名，protected则会加上*作为前缀
- **offset**：这个值记录的就是上面说的通过数组保存的属性值的索引，也就是说属性值保存在一个数组中，然后将其在数组中的位置保存在offset中，另外需要说明的一点的是普通属性、静态属性这个值用法是不一样的，静态属性是类的范畴，与对象无关，所以其offset为default*staticmemberstable*数组的下标：*0,*、*1*、*2.....*，而普通属性归属于对象，每个对象有其各自的属性，所以这个*offset*记录的实际是 _各属性在*object*中偏移值 (在后面《3.4.2 对象》一节我们再具体说明普通属性的存储方式)，其值是：40、56、72......是按照zval的内存大小偏移的
- **flags**：bit位，标识的是属性的信息，如public、private、protected及是否为静态属性

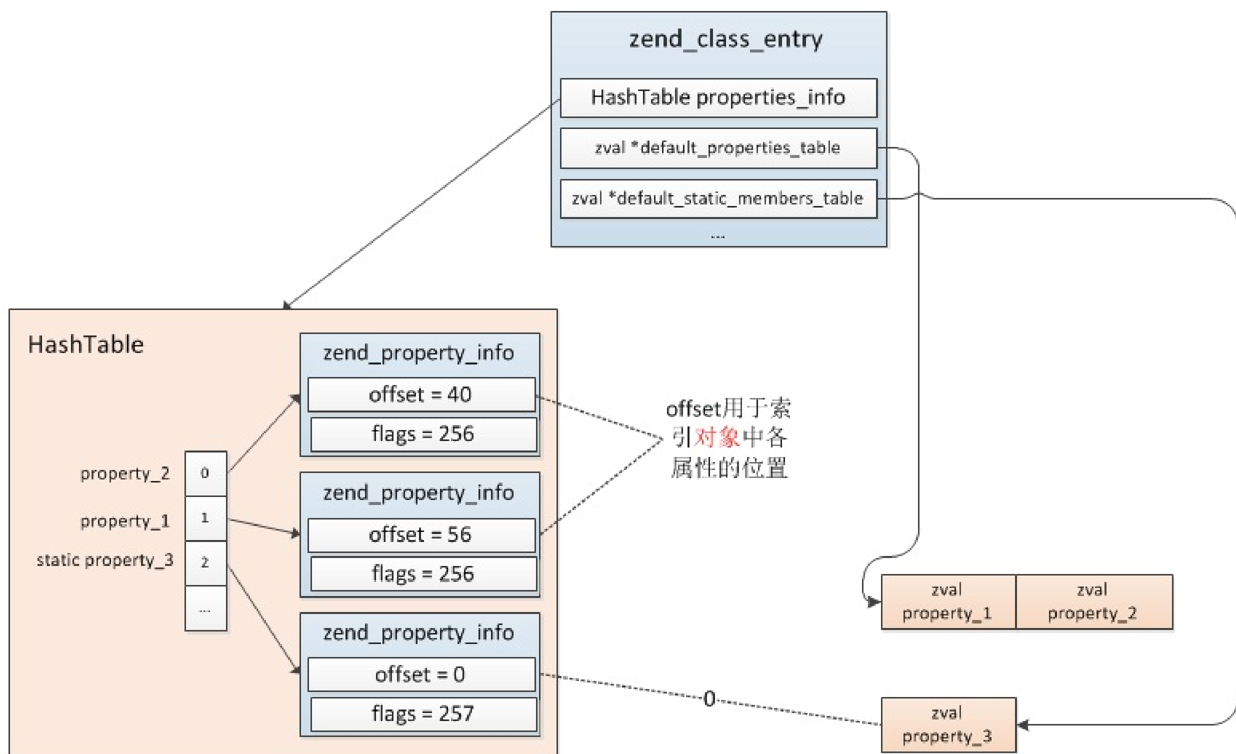
所以访问成员属性时首先是根据属性名查找到此属性的存储位置，然后再进一步获取属性值。

举个例子：

```
class my_class {
    public $property_1 = "aa";
    public $property_2 = array();

    public static $property_3 = 110;
}
```

则 **default_properties_table**、**default_static_properties_table**、**properties_info** 关系图：

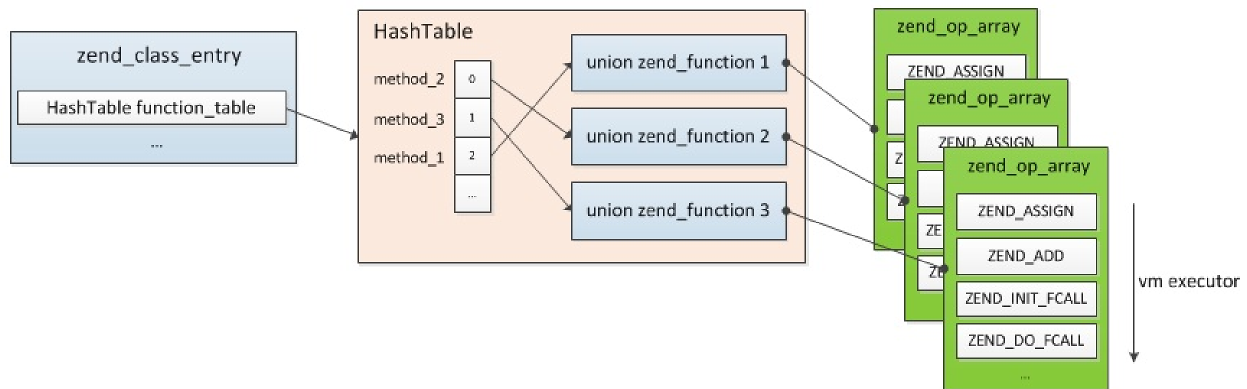


下面我们再看下普通成员属性与静态成员属性的不同：静态成员变量保存在类中，各对象共享同一份数据，而普通属性属于对象，各对象独享。

成员属性在类编译阶段就已经分配了zval，静态与普通的区别在于普通属性在创建一个对象时还会重新分配zval（这个过程类似zend引擎执行前分配在zend_execute_data后面的动态变量空间），对象对普通属性的操作都是在其自己的空间进行的，各对象隔离，而静态属性的操作始终是在类的空间内，各对象共享。

3.4.1.4 成员方法

每个类可以定义若干属于本类的函数(称之为成员方法)，这种函数与普通的function相同，只是以类的维度进行管理，不是全局性的，所以成员方法保存在类中而不是EG(function_table)。



> 成员方法的定义：
>
> 【修饰符(public/private/protected/static/abstruct/final)】function 【&】【成员方法名】(【参数列表】)【返回值类型】{【成员方法】};

成员方法也有静态、非静态之分，静态方法中不能使用$this，因为其操作的作用域全部都是类的而不是对象的，而非静态方法中可以通过$this访问属于本对象的成员属性。

静态方法也是通过static关键词定义：

```
class my_class {
    static public function test() {
        $a = "hi~";
        echo $a;
    }
}
//静态方法可以这么调用：
my_class::test();

//也可以这样：
$method = 'test';
my_class::$method();
```

静态方法中调用其它静态方法或静态变量可以通过 **self** 访问。

成员方法的调用与普通function过程基本相同，根据对象所属类或直接根据类取到method的zend_function，然后执行，具体的过程《3.3 Zend引擎执行过程》已经详细说过，这里不再重复。

3.4.1.5 自定义类的编译

前面我们先介绍了类的相关组成部分，接下来我们从一个例子简单看下类的编译过程，这个过程最终的产物就是zend_class_entry。

```
//示例
class Human {
    public $aa = array(1,2,3);
}

class User extends Human
{
    const type = 110;

    static $name = "uuu";
    public $uid = 900;
    public $sex = 'w';

    public function __construct(){
    }

    public function getName(){
        return $this->name;
    }
}
```

> 类的定义组成部分：
>
> 【修饰符(abstract/final)】 class 【类名】 【extends 父类】 【implements 接口1,接口2】 {}

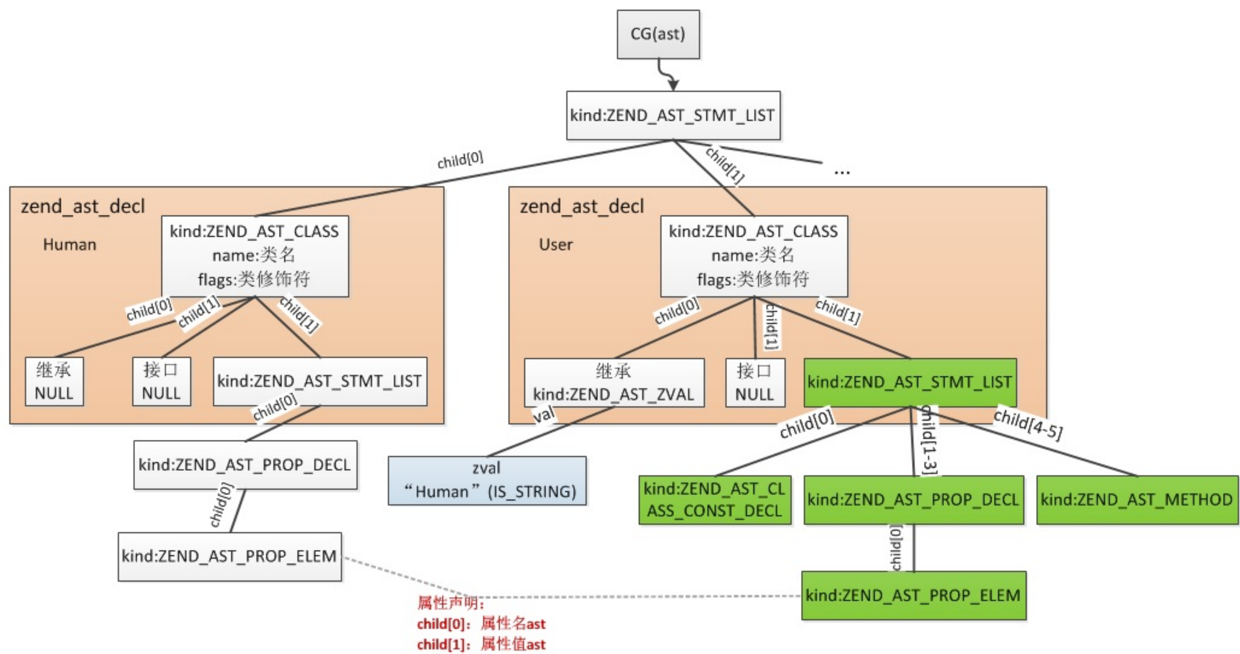
语法规则为：

```
class_declaration_statement:
        class_modifiers T_CLASS { $<num>$ = CG(zend_lineno); }
        T_STRING extends_from implements_list backup_doc_comment
'{' class_statement_list '}'
            { $$ = zend_ast_create_decl(ZEND_AST_CLASS, $1, $<nu
m>3, $7, zend_ast_get_str($4), $5, $6, $9, NULL); }
    |   T_CLASS { $<num>$ = CG(zend_lineno); }
        T_STRING extends_from implements_list backup_doc_comment
'{' class_statement_list '}'
            { $$ = zend_ast_create_decl(ZEND_AST_CLASS, 0, $<num>
2, $6, zend_ast_get_str($3), $4, $5, $8, NULL); }
;

//整个类内为list，每个成员属性、成员方法都是一个子节点
class_statement_list:
        class_statement_list class_statement
            { $$ = zend_ast_list_add($1, $2); }
    |   /* empty */
            { $$ = zend_ast_create_list(0, ZEND_AST_STMT_LIST);
}
;

//类内语法规则：成员属性、成员方法
class_statement:
        variable_modifiers property_list ';'
            { $$ = $2; $$->attr = $1; }
    |   T_CONST class_const_list ';'
            { $$ = $2; RESET_DOC_COMMENT(); }
    |   T_USE name_list trait_adaptations
            { $$ = zend_ast_create(ZEND_AST_USE_TRAIT, $2, $3);
}
    |   method_modifiers function returns_ref identifier backup_
doc_comment '(' parameter_list ')'
        return_type method_body
            { $$ = zend_ast_create_decl(ZEND_AST_METHOD, $3 | $1
, $2, $5,
                  zend_ast_get_str($4), $7, NULL, $10, $9); }
;
```

生成的抽象语法树：



类的语法树根节点为ZEND_AST_CLASS，此节点有3个子节点：继承子节点、实现接口子节点、类中声明表达式节点，其中child2为zend_ast_list，每个常量定义、成员属性、成员方法对应一个节点，比如上面的例子中user类有6个子节点，这些子节点类型有3类：常量声明(ZEND_AST_CLASS_CONST_DECL)、属性声明(ZEND_AST_PROP_DECL)、方法声明(ZEND_AST_METHOD)。

编译为opcodes操作为： `zend_compile_class_decl()` ，它的输入就是ZEND_AST_CLASS节点，这个函数中再针对常量、属性、方法、继承、接口等分别处理。

```
void zend_compile_class_decl(zend_ast *ast)
{
    zend_ast_decl *decl = (zend_ast_decl *) ast;
    zend_ast *extends_ast = decl->child[0]; //继承类节点，zen_ast_zval节点，存的是父类名
    zend_ast *implements_ast = decl->child[1]; //实现接口节点
    zend_ast *stmt_ast = decl->child[2]; //类中声明的常量、属性、方法

    zend_string *name, *lcname;
    zend_class_entry *ce = zend_arena_alloc(&CG(arena), sizeof(zend_class_entry));
    zend_op *opline;
    ...
```

```
    lcname = zend_new_interned_string(lcname);

    ce->type = ZEND_USER_CLASS; //类型为用户自定义类
    ce->name = name; //类名
    zend_initialize_class_data(ce, 1);
    ...
    if (extends_ast) {
        ...
        //有继承的父类则首先生成一条ZEND_FETCH_CLASS的opcode
        zend_compile_class_ref(&extends_node, extends_ast, 0);
    }

    //在当前父空间生成一条opcode
    opline = get_next_op(CG(active_op_array));
    zend_make_var_result(&declare_node, opline);
    ...
    opline->op2_type = IS_CONST;
    LITERAL_STR(opline->op2, lcname);

    if (decl->flags & ZEND_ACC_ANON_CLASS) {
        //暂不清楚这种情况
    }else{
        zend_string *key;

        if (extends_ast) {
            opline->opcode = ZEND_DECLARE_INHERITED_CLASS; //有继
承的类为这个opcode
            opline->extended_value = extends_node.u.op.var;
        } else {
            opline->opcode = ZEND_DECLARE_CLASS; //无继承的类为这个
opcode
        }

        key = zend_build_runtime_definition_key(lcname, decl->le
x_pos); //这个key并不是类名，而是：类名+file+lex_pos

        opline->op1_type = IS_CONST;
        LITERAL_STR(opline->op1, key);//将这个临时key保存到操作数1中
```

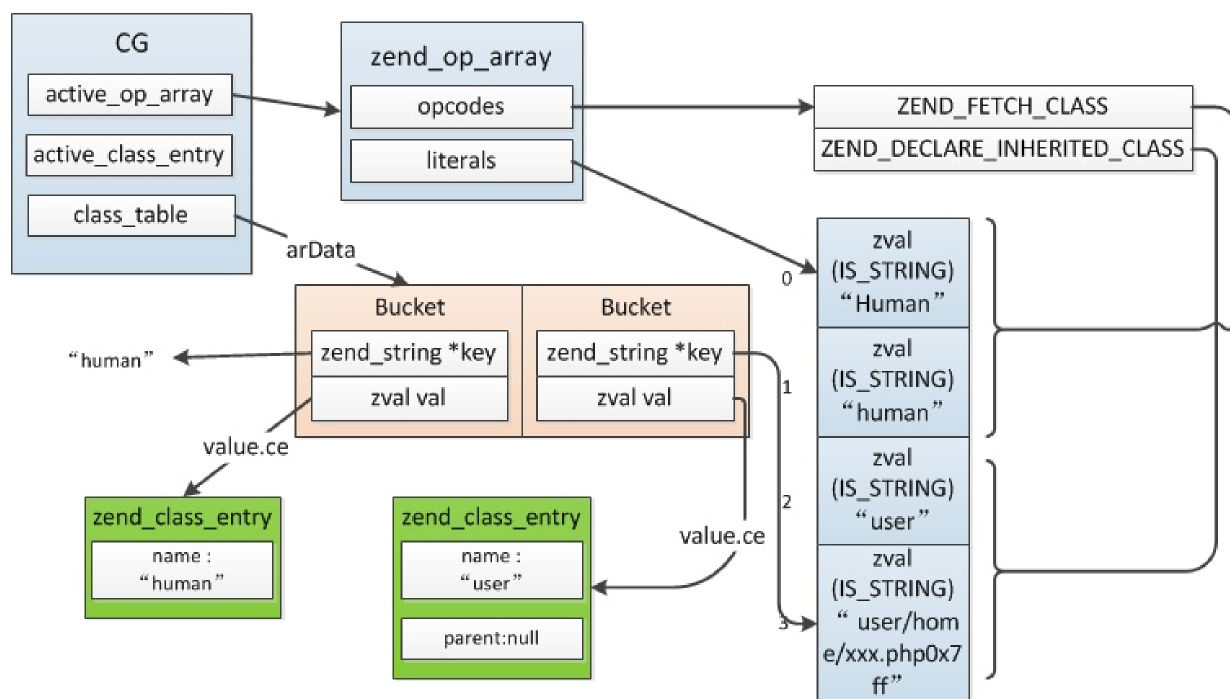
```
        zend_hash_update_ptr(CG(class_table), key, ce); //将半成品的zend_class_entry插入CG(class_table)，注意这里并不是执行时用于索引类的，它的key不是类名!!!
    }
    CG(active_class_entry) = ce;
    zend_compile_stmt(stmt_ast); //将常量、成员属性、方法编译到CG(active_class_entry)中

    ...

    CG(active_class_entry) = original_ce;
}
```

上面这个过程主要操作是新分配一个zend_class_entry，如果有继承的话首先生成一条ZEND_FETCH_CLASS的opcode，然后生成一条类声明的opcode（这个地方与之前3.2.1.3节介绍函数的编译时相同），接着就是编译常量、属性、成员方法到新分配的zend_class_entry中，这个过程还有一个容易误解的地方：将生成的zend_class_entry插入到CG(class_table)哈希表中，这个操作这是中间步骤，它的key并不是类名，而是类名后面带来一长串其它的字符，也就是这个时候通过类名在class_table是索引不到对应类的，后面我们会说明这样处理的作用。

Human类情况比较简单，不再展开，我们看下User类在 `zend_compile_class_decl()` 中执行到 `zend_compile_stmt(stmt_ast)` 这步时关键数据结构：



接下来我们分别看下常量、成员属性、方法的编译过程。

(1)常量编译

常量的节点类型为：`ZEND_AST_CLASS_CONST_DECL`，每个常量对应一个这样的节点，处理函数为: `zend_compile_class_const_decl()`：

```
void zend_compile_class_const_decl(zend_ast *ast)
{
    zend_ast_list *list = zend_ast_get_list(ast);
    zend_class_entry *ce = CG(active_class_entry);
    uint32_t i;

    for (i = 0; i < list->children; ++i) { //const声明了多个常量，遍历编译每个子节点
        zend_ast *const_ast = list->child[i];
        zend_ast *name_ast = const_ast->child[0]; //常量名节点
        zend_ast *value_ast = const_ast->child[1];//常量值节点
        zend_string *name = zend_ast_get_str(name_ast); //常量名
        zval value_zv;

        //取出常量值
        zend_const_expr_to_zval(&value_zv, value_ast);

        name = zend_new_interned_string_safe(name);
        //将常量添加到zend_class_entry.constants_table哈希表中
        if (zend_hash_add(&ce->constants_table, name, &value_zv) == NULL) {
            ...
        }
        ...
    }
}
```

(2)属性编译

属性节点类型为: `ZEND_AST_PROP_DECL` ，对应的处理函数: `zend_compile_prop_decl()` :

```
void zend_compile_prop_decl(zend_ast *ast)
{
    zend_ast_list *list = zend_ast_get_list(ast);
    uint32_t flags = list->attr; //属性修饰符：static、public、priv
ate、protected
    zend_class_entry *ce = CG(active_class_entry);
    uint32_t i, children = list->children;

    for (i = 0; i < children; ++i) {
        zend_ast *prop_ast = list->child[i]; //这个节点类型为：ZEND
_AST_PROP_ELEM
        zend_ast *name_ast = prop_ast->child[0]; //属性名节点
        zend_ast *value_ast = prop_ast->child[1]; //属性值节点
        zend_ast *doc_comment_ast = prop_ast->child[2];
        zend_string *name = zend_ast_get_str(name_ast); //属性名
        zend_string *doc_comment = NULL;
        zval value_zv;
        ...
        //检查该属性是否在当前类中已经定义
        if (zend_hash_exists(&ce->properties_info, name)) {
            zend_error_noreturn(...);
        }
        if (value_ast) {
            //取出默认值
            zend_const_expr_to_zval(&value_zv, value_ast);
        } else {
            //默认值为null
            ZVAL_NULL(&value_zv);
        }

        name = zend_new_interned_string_safe(name);
        //保存属性
        zend_declare_property_ex(ce, name, &value_zv, flags, doc
_comment);
    }
}
```

开始的时候我们已经介绍：属性值是通过 数组 保存的，然后其存储位置通过以 属性名 为key的哈希表保存，使用的时候先从这个哈希表中找到属性信息同时得到属性值的保存位置，然后再进一步取出属性值。

`zend_declare_property_ex()` 这步操作就是来确定属性的存储位置的，它将属性值按静态、非静态分别保存在default_static_members_table、default_properties_table两个数组中，同时将其存储位置保存到属性结构的offset中。

```
//zend_API.c
ZEND_API int zend_declare_property_ex(zend_class_entry *ce, zend_string *name, zval *property, int access_type,...)
{
    zend_property_info *property_info, *property_info_ptr;

    if (ce->type == ZEND_INTERNAL_CLASS) {//内部类
        ...
    }else{
        property_info = zend_arena_alloc(&CG(arena), sizeof(zend_property_info));
    }

    if (access_type & ZEND_ACC_STATIC) {
        //静态属性
        ...
        property_info->offset = ce->default_static_members_count++; //分配属性编号，同变量一样，静态属性的就是数组索引
        ce->default_static_members_table = perealloc(ce->default_static_members_table, sizeof(zval) * ce->default_static_members_count, ce->type == ZEND_INTERNAL_CLASS);

        ZVAL_COPY_VALUE(&ce->default_static_members_table[property_info->offset], property);
        if (ce->type == ZEND_USER_CLASS) {
            ce->static_members_table = ce->default_static_members_table;
        }
    }else{
        //非静态属性
```

```
        ...
        //非静态属性值存储在对象中，所以与静态属性不同，它的offset并不是d
efault_properties_table数组索引
        //而是相对于zend_object大小的(因为普通属性值数组保存在zend_obje
ct结构之后，这个与局部变量、zend_execute_data关系一样)
        property_info->offset = OBJ_PROP_TO_OFFSET(ce->default_p
roperties_count);
        ce->default_properties_count++;
        ce->default_properties_table = perealloc(ce->default_pro
perties_table, sizeof(zval) * ce->default_properties_count, ce->
type == ZEND_INTERNAL_CLASS);

        ZVAL_COPY_VALUE(&ce->default_properties_table[OBJ_PROP_T
O_NUM(property_info->offset)], property);
    }

    //设置property_info其它的一些值
    ...
}
```

这个操作中重点是offset的计算方式，静态属性这个比较好理解，就是default_static_members_table数组索引；非静态属性zend_class_entry.default_properties_table保存的只是默认属性值，我们在下一篇介绍对象时再具体说明object、class之间属性的存储关系。

(3)成员方法编译 3.4.1.4一节已经介绍过成员方法与普通函数的关系，两者没有很大的区别，实现上是相同，不同的地方在于成员方法保存在各zend_class_entry中，调用时会有一些可见性方面的限制，如private、public、protected，还有一些专有用法，比如this、self等，但在编译、执行、存储结构等方面两者基本是一致的。

成员方法的语法树根节点为 `ZEND_AST_METHOD` ：

```
void zend_compile_stmt(zend_ast *ast)
{
    ...
    switch (ast->kind) {
        ...
        case ZEND_AST_FUNC_DECL: //函数
        case ZEND_AST_METHOD:  //成员方法
            zend_compile_func_decl(NULL, ast);
            break;
        ...
    }
}
```

如果你还记得3.2.1.3函数处理的过程就会发现函数、成员方法的编译是同一个函数： `zend_compile_func_decl()` 。

```
void zend_compile_func_decl(znode *result, zend_ast *ast)
{
    //参数、函数内语法编译等不看了，与函数的相同，不清楚请看3.2.1.3节
    ...

    if (is_method) {
        zend_bool has_body = stmt_ast != NULL;
        zend_begin_method_decl(op_array, decl->name, has_body);
    } else {
        //函数是在当前空间生成了一条ZEND_DECLARE_FUNCTION的opcode
        //然后在zend_do_early_binding()中"执行"了这条opcode，即将函数添加到CG(function_table)
        zend_begin_func_decl(result, op_array, decl);
    }
    ...
}
```

这个过程之前已经说过，这里不再重复，我们只看下与普通函数处理不同的地方： `zend_begin_method_decl()` ，它的工作也比较简单，最重要的一个地方就是将成员方法的zend*oparray插入 __zend_class_entry.function_table*。

```
void zend_begin_method_decl(zend_op_array *op_array, zend_string
 *name, zend_bool has_body)
{
    zend_class_entry *ce = CG(active_class_entry);
    ...

    op_array->scope = ce;
    op_array->function_name = zend_string_copy(name);

    lcname = zend_string_tolower(name);
    lcname = zend_new_interned_string(lcname);

    //插入类的function_table中
    if (zend_hash_add_ptr(&ce->function_table, lcname, op_array)
 == NULL) {
        zend_error_noreturn(..);
    }

    //后面主要是设置一些构造函数、析构函数、魔术方法指针，以及其它一些可见性
、静态非静态的检查
    ...
}
```

上面我们分别介绍了常量、成员属性、方法的编译过程，最后再用一张图总结下整个类的编译过程：

图中还有一步我们没有说到：**zend_do_early_binding()** ，这是非常关键的一步，如果你看过3.2.1.3一节那么对这个函数应该不陌生，没错，在函数编译的最后一步也会调用这个函数，它的作用是将编译的function以函数名为key添加到CG(function_table)中，同样地上面整个过程中你可能发现所有的操作都是针对zend_class_entry，并没有发现最后把它存到什么位置了，这最后的一步就是把zend_class_entry以类名为key添加到CG(class_table)。

```
void zend_do_early_binding(void)
{
    ...
    switch (opline->opcode) {
        ...
        case ZEND_DECLARE_CLASS:
            if (do_bind_class(CG(active_op_array), opline, CG(cl
ass_table), 1) == NULL) {
                return;
            }
            table = CG(class_table);
            break;
        case ZEND_DECLARE_INHERITED_CLASS:
            //比较长，后面单独摘出来
            break;
    }

    //将那个以(类名+file+lex_pos)为key的值从CG(class_table)中删除
    //同时删除两个相关的literals：key、类名
    zend_hash_del(table, Z_STR_P(CT_CONSTANT(opline->op1)));
    zend_del_literal(CG(active_op_array), opline->op1.constant);
    zend_del_literal(CG(active_op_array), opline->op2.constant);
    MAKE_NOP(opline); //将ZEND_DECLARE_CLASS或ZEND_DECLARE_INHERI
TED_CLASS的opcode置为空，表示已执行
}
```

这个地方会有两种情况，上面我们说过，如果是普通的没有继承的类定义会生成一条 `ZEND_DECLARE_CLASS` 的opcode，而有继承的类则会生成 `ZEND_FETCH_CLASS` 、 `ZEND_DECLARE_INHERITED_CLASS` 两条opcode，这两种有很大的不同，接下来我们具体看下：

> **(1)**无继承类：这种情况直接调用`do_bind_class()`处理了。```c ZEND_API zend_class_entry *do_bind_class(const zend_op_array* op_array, const zend_op *opline, HashTable* class_table, zend_bool compile_time) { if (compile_time) { //编译时 //还记得zend_compile_class_decl()中有一个把zend_class_entry以(类名+file+lex_pos) //为key存入CG(class_table)的操作吗？那个key的存储位置保存在op1中了 //这里就是从op_array.literals中取出那个key op1 = CT_CONSTANT_EX(op_array, opline->op1.constant); //op2为类名 op2 = CT_CONSTANT_EX(op_array, opline->op2.constant); } else { //运行时，如果当前类在编译阶段没有编译完成则也有可能在zend_execute执行阶段完成 op1 = RT_CONSTANT(op_array, opline->op1); op2 = RT_CONSTANT(op_array, opline->op2); } //从CG(class_table)中取出zend_class_entry if ((ce = zend_hash_find_ptr(class_table, Z_STR_P(op1))) == NULL) { zend_error_noreturn(E_COMPILE_ERROR, ...); return NULL; } ce->refcount++; //这里加1是因为CG(class_table)中多了一个bucket指向这个ce了

```
//以标准类名为key将zend_class_entry插入CG(class_table)
//这才是后面要用到的类
if (zend_hash_add_ptr(class_table, Z_STR_P(op2), ce) == NULL) {
    //插入失败
    return NULL;
}else{
    //插入成功
    return ce;
}
```

}

```
> 这个函数就是将类以 __正确的类名__ 为key插入到CG(class_table)，这一步完成后`zend_do_early_binding()`后面就将`ZEND_DECLARE_CLASS`这条opcode置为0了，这样在运行时就直接跳过此opcode了，现在清楚为什么执行时会有很多为0的opcode了吧？

> __(2)有继承类：__ 这种类是有继承的父类，它的定义有两条opcode：`ZEND_FETCH_CLASS`、`ZEND_DECLARE_INHERITED_CLASS`，上面我们一张图画过示例中user类编译的情况，我们先看下它的opcode再作说明。
```

````
![](../img/ast_fetch_class.png)

```c
case ZEND_DECLARE_INHERITED_CLASS:
{
    zend_op *fetch_class_opline = opline-1;
    zval *parent_name;
    zend_class_entry *ce;

    parent_name = CT_CONSTANT(fetch_class_opline->op2); //父类名

    //在EG(class_table)中查找父类(注意：EG(class_table)与CG(class_table)指向同一个位置)
    if (((ce = zend_lookup_class_ex(Z_STR_P(parent_name), parent_name + 1, 0)) == NULL) || ...) {
        //没找到父类，有可能父类没有定义、有可能父类在子类之后定义的......
        if (CG(compiler_options) & ZEND_COMPILE_DELAYED_BINDING) {
            ...
            //将opcode重置为ZEND_DECLARE_INHERITED_CLASS_DELAYED
            opline->opcode = ZEND_DECLARE_INHERITED_CLASS_DELAYED;

            opline->result_type = IS_UNUSED;
            opline->result.opline_num = -1;
        }
        return;
    }
    //注册继承类
    if (do_bind_inherited_class(CG(active_op_array), opline, CG(class_table), ce, 1) == NULL) {
        return;
    }

    //清理无用的opcode：ZEND_FETCH_CLASS，重置为0，执行时直接跳过
    zend_del_literal(CG(active_op_array), fetch_class_opline->op2.constant);
    MAKE_NOP(fetch_class_opline);

    table = CG(class_table);
    break;
````
````

```
}
```

> 通过上面的处理我们可以看到，首先是查找父类：
>
> > 1)如果父类没有找到则将opcode置为`ZEND_DECLARE_INHERITED_CLASS_DELAYED`，这种情况下当前类是没有编译到CG(class_table)中去的，也就是这个时候这个类是无法使用的，在执行的时候会再次尝试这个过程，那个时候如果找到父类了则再加入EG(class_table)；
> >
> > 2)如果找到父类了则与无继承的类处理一样，将zend_class_entry添加到CG(class_table)中，然后将对应的两条opcode删掉，除了这个外还有一个非常重要的操作：`zend_do_inheritance()`，这里主要是进行属性、常量、成员方法的合并、拷贝，这个过程这里暂不展开，《3.4.3继承》一节再作具体说明。

总结：

上面我们介绍了类的编译过程，整个流程东西比较但并不复杂，主要围绕zend_class_entry进行的操作，另外我们知道了类插入EG(class_table)的过程，这个相当于类的声明在编译阶段提前"执行"了，也有可能因为父类找不到等原因延至运行时执行，清楚了这个过程你应该能明白下面这些例子为什么有的可以运行而有的则报错的原因了吧？

```
//情况1
new A();

class A extends B{}
class B{}

===================
完整opcodes：
1 ZEND_NEW                    => 执行到这报错，因为此时A因为找不到B尚未编译进EG(class_table)
2 ZEND_DO_FCALL
3 ZEND_FETCH_CLASS
4 ZEND_DECLARE_INHERITED_CLASS
5 ZEND_DECLARE_CLASS           => 注册class B
6 ZEND_RETURN

实际执行顺序：5->1->2->3->4->6
```

```
//情况2
class A extends B{}
class B{}

new A();
===================
完整opcodes：
1 ZEND_FETCH_CLASS
2 ZEND_DECLARE_INHERITED_CLASS => 注册class A，此时已经可以找到B
3 ZEND_DECLARE_CLASS           => 注册class B
4 ZEND_NEW
5 ZEND_DO_FCALL
6 ZEND_RETURN

实际执行顺序：3->1->2->4->5->6，执行到4时A都已经注册，所以可以执行
```

```
//情况3
class A extends B{}
class B extends C{}
class C{}

new A();
===================
完整opcodes：
1 ZEND_FETCH_CLASS              => 找不到B,直接报错
2 ZEND_DECLARE_INHERITED_CLASS
3 ZEND_FETCH_CLASS
4 ZEND_DECLARE_INHERITED_CLASS => 注册class B，此时可以找到C，所以注册成功
5 ZEND_DECLARE_CLASS            => 注册class C
6 ZEND_NEW
7 ZEND_DO_FCALL
8 ZEND_RETURN

实际执行顺序：5->1->2->3->4->5->6->7->8，执行到1发现还是找不到父类B，报错
```

3.4.1.6 内部类

前面我们介绍了类的基本组成以及用户自定义类的编译，除了在PHP代码中可以定义一个类，我们也可以在内核或扩展中定义一个类(与定义内部函数类似)，这种类称之为 内部类。

相比于用户自定义类的编译实现，内部类的定义比较简单，也更加灵活，可以进行一些个性化的处理，比如我们可以定义创建对象的钩子函数： `create_object` ，从而在对象实例化时调用我们自己定义的函数完成，这样我们就可以进行很多其它的操作。

内部类的定义简单的概括就是 `创建一个zend_class_entry结构，然后插入到EG(class_table)中` ，涉及的操作主要有：

- 注册类到符号表
- 实现继承、接口
- 定义常量
- 定义成员属性

- 定义成员方法

实际这些与用户自定义类的实现相同，只是内部类直接调用相关API完成这些操作，具体的API接口本节不再介绍，我们将在后面介绍扩展开发一章中再系统说明。

3.4.2 对象

对象是类的实例，PHP中要创建一个类的实例，必须使用 new 关键字。类应在被实例化之前定义（某些情况下则必须这样，比如3.4.1最后那几个例子）。

3.4.2.1 对象的数据结构

对象的数据结构非常简单：

```
typedef struct _zend_object     zend_object;

struct _zend_object {
    zend_refcounted_h gc; //引用计数
    uint32_t          handle;
    zend_class_entry *ce; //所属类
    const zend_object_handlers *handlers; //对象操作处理函数
    HashTable        *properties;
    zval              properties_table[1]; //普通属性值数组
};
```

几个主要的成员：

(1)handle: 一次request期间对象的编号，每个对象都有一个唯一的编号，与创建先后顺序有关，主要在垃圾回收时用，下面会详细说明。

(2)ce: 所属类的zend_class_entry。

(3)handlers: 这个保存的对象相关操作的一些函数指针，比如成员属性的读写、成员方法的获取、对象的销毁/克隆等等，这些操作接口都有默认的函数。

```
struct _zend_object_handlers {
    int                                     offset;
    zend_object_free_obj_t                  free_obj; //释放对象
    zend_object_dtor_obj_t                  dtor_obj; //销毁对象
    zend_object_clone_obj_t                 clone_obj;//复制对象

    zend_object_read_property_t             read_property; //读取成员属性
```

```
    zend_object_write_property_t            write_property;//修改
成员属性
    ...
}

//默认值处理handler
ZEND_API zend_object_handlers std_object_handlers = {
    0,
    zend_object_std_dtor,                   /* free_obj */
    zend_objects_destroy_object,            /* dtor_obj */
    zend_objects_clone_obj,                 /* clone_obj */
    zend_std_read_property,                 /* read_property */
    zend_std_write_property,                /* write_property */
    zend_std_read_dimension,                /* read_dimension */
    zend_std_write_dimension,               /* write_dimension */

    zend_std_get_property_ptr_ptr,          /* get_property_ptr_
ptr */
    NULL,                                   /* get */
    NULL,                                   /* set */
    zend_std_has_property,                  /* has_property */
    zend_std_unset_property,                /* unset_property */
    zend_std_has_dimension,                 /* has_dimension */
    zend_std_unset_dimension,               /* unset_dimension */

    zend_std_get_properties,                /* get_properties */
    zend_std_get_method,                    /* get_method */
    NULL,                                   /* call_method */
    zend_std_get_constructor,               /* get_constructor */

    zend_std_object_get_class_name,         /* get_class_name */
    zend_std_compare_objects,               /* compare_objects */

    zend_std_cast_object_tostring,          /* cast_object */
    NULL,                                   /* count_elements */
    zend_std_get_debug_info,                /* get_debug_info */
    zend_std_get_closure,                   /* get_closure */
    zend_std_get_gc,                        /* get_gc */
    NULL,                                   /* do_operation */
    NULL,                                   /* compare */
```

```
}
```

> **Note:** 这些handler用于操作对象(如：设置、读取属性)，std_object_handlers是PHP定义的默认、标准的处理函数，在扩展中可以自定义handler，比如：重定义write_property，这样设置一个对象的属性时将调用扩展自己定义的处理函数，让扩展拥有了更高的控制权限。
>
> 需要注意的是：const zend_object_handlers *handlers*，这里的*handlers*指针加了*const*修饰符，*const*修饰的是*handlers**指向的对象，而不是handlers指针本身，所以扩展中可以将一个对象的handlers修改为另一个zend_object_handlers指针，但无法修改zend_object_handlers中的值，比如：`obj->handlers->write_property = xxx` 将报错，而：`obj->handlers = xxx` 则是可以的。

(4)properties: 普通成员属性哈希表，对象创建之初这个值为NULL，主要是在动态定义属性时会用到，与properties_table有一定关系，下一节我们将单独说明，这里暂时忽略。

(5)properties_table: 成员属性数组，还记得我们在介绍类一节时提过非静态属性存储在对象结构中吗？就是这个properties_table！注意，它是一个数组，`zend_object` 是个变长结构体，分配时会根据非静态属性的数量确定其大小。

3.4.2.2 对象的创建

PHP中通过 `new + 类名` 创建一个类的实例，我们从一个例子分析下对象创建的过程中都有哪些操作。

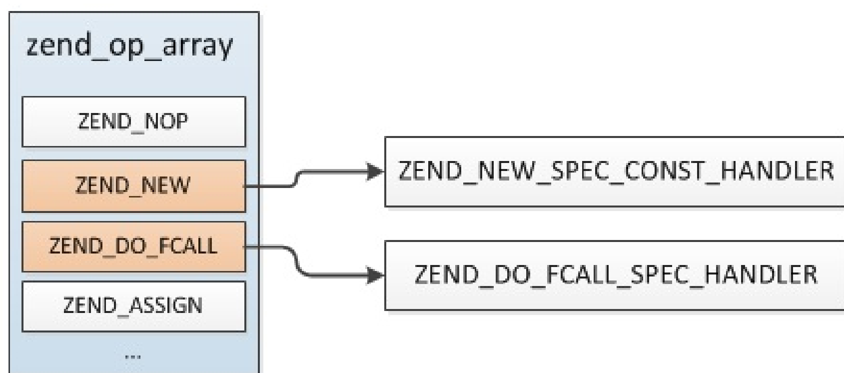
```
class my_class
{
    const TYPE = 90;
    public $name = "pangudashu";
    public $ids = array();
}

$obj = new my_class();
```

类的定义就不用再说了，我们只看 `$obj = new my_class();` 这一句，这条语句包括两部分：实例化类、赋值，下面看下实例化类的语法规则：

```
new_expr:
        T_NEW class_name_reference ctor_arguments
            { $$ = zend_ast_create(ZEND_AST_NEW, $2, $3); }
    |   T_NEW anonymous_class
            { $$ = $2; }
;
```

从语法规则可以很直观的看出此语法的两个主要部分：类名、参数列表，编译器在解析到实例化类时就创建一个 `ZEND_AST_NEW` 类型的节点，后面编译为opcodes的过程我们不再细究，这里直接看下最终生成的opcodes。



你会发现实例化类产生了两条opcode(实际可能还会更多)：ZEND_NEW、ZEND_DO_FCALL，除了创建对象的操作还有一条函数调用的，没错，那条就是调用 `构造方法` 的操作。

根据opcode、操作数类型可知 `ZEND_NEW` 对应的处理handler为 `ZEND_NEW_SPEC_CONST_HANDLER()` ：

```
static int ZEND_NEW_SPEC_CONST_HANDLER(zend_execute_data *execut
e_data)
{
    zval object_zval;
    zend_function *constructor;
    zend_class_entry *ce;
    ...
    //第1步：根据类名查找zend_class_entry
    ce = zend_fetch_class_by_name(Z_STR_P(EX_CONSTANT(opline->op
1)), ...);
    ...
    //第2步：创建&初始化一个这个类的对象
    if (UNEXPECTED(object_init_ex(&object_zval, ce) != SUCCESS))
 {
        HANDLE_EXCEPTION();
    }
    //第3步：获取构造方法
    //获取构造方法函数，实际就是直接取zend_class_entry.constructor
    //get_constructor => zend_std_get_constructor()
    constructor = Z_OBJ_HT(object_zval)->get_constructor(Z_OBJ(o
bject_zval));

    if (constructor == NULL) {
        ...
        //此opcode之后还有传参、调用构造方法的操作
        //所以如果没有定义构造方法则直接跳过这些操作
        ZEND_VM_JMP(OP_JMP_ADDR(opline, opline->op2));
    }else{
        //定义了构造方法
        //初始化调用构造函数的zend_execute_data
        zend_execute_data *call = zend_vm_stack_push_call_frame(
...);
        call->prev_execute_data = EX(call);
        EX(call) = call;
        ...
    }
}
```

从上面的创建对象的过程看整个流程主要分为三步：首先是根据类名在EG(class_table)中查找对应zend_class_entry、然后是创建并初始化一个对象、最后是初始化调用构造函数的zend_execute_data。

我们再具体看下第2步创建、初始化对象的操作，`object_init_ex(&object_zval, ce)` 最终调用的是 `_object_and_properties_init()` 。

```
//zend_API.c
ZEND_API int _object_and_properties_init(zval *arg, zend_class_entry *class_type, ...)
{
    //检查类是否可以实例化
    ...

    //用户自定义的类create_object都是NULL
    //只有PHP几个内部的类有这个值，比如exception、error等
    if (class_type->create_object == NULL) {
        //分配一个对象
        ZVAL_OBJ(arg, zend_objects_new(class_type));
        ...
        //初始化成员属性
        object_properties_init(Z_OBJ_P(arg), class_type);
    } else {
        //调用自定义的创建object的钩子函数
        ZVAL_OBJ(arg, class_type->create_object(class_type));
    }
    return SUCCESS;
}
```

还记得上一节介绍zend_class_entry时有几个自定义的钩子函数吗？如果定义了 `create_object` 这个地方就会调用自定义的函数来创建zend_object，这种情况通常发生在内核或扩展中定义的内部类(当然用户自定义类也可以修改，但一般不会那样做)；用户自定义类在这个地方又具体分了两步：分配对象结构、初始化成员属性，我们继续看下这里面的处理。

(1)分配对象结构:zend_object

```
//zend_objects.c
ZEND_API zend_object *zend_objects_new(zend_class_entry *ce)
{
    //分配zend_object
    zend_object *object = emalloc(sizeof(zend_object) + zend_obj
ect_properties_size(ce));

    zend_object_std_init(object, ce);
    //设置对象的操作handler为std_object_handlers
    object->handlers = &std_object_handlers;
    return object;
}
```

有个地方这里需要特别注意：分配对象结构的内存并不仅仅是zend*object*的大小。我们在*3.4.2.1*介绍*propertiestable*时说过这是一个变长数组，它用来存放非静态属性的值，所以分配*zendobject*时需要加上非静态属性所占用的内存大小： `zend_object_properties_size()` ，根据普通非静态属性个数确定，如果没有定义***get()***、*set()*等魔术方法则占用内存就是*:* _属性数**sizeof(zval)* ，如果定义了这些魔术方法那么会多分配一个zval的空间，这个多出来zval的用途下面介绍成员属性的读写时再作说明。

另外这里还有一个关键操作：将**object**编号并插入**EG(objects_store).object_buckets**数组。zend_object有个成员：handle，这个值在一次request期间所有实例化对象的编号，每调用 `zend_objects_new()` 实例化一个对象就会将其插入到object_buckets数组中，其在数组中的下标就是handle。这个过程是在 `zend_objects_store_put()` 中完成的。

```
//zend_objects_API.c
ZEND_API void zend_objects_store_put(zend_object *object)
{
    int handle;

    if (EG(objects_store).free_list_head != -1) {
        //这种情况主要是gc中会将中间一些object销毁，空出一些bucket位置
        //然后free_list_head就指向了第一个可用的bucket位置
        //后面可用的保存在第一个空闲bucket的handle中
        handle = EG(objects_store).free_list_head;
        EG(objects_store).free_list_head = GET_OBJ_BUCKET_NUMBER
(EG(objects_store).object_buckets[handle]);
    } else {
        if (EG(objects_store).top == EG(objects_store).size) {
            //扩容
        }
        //递增加1
        handle = EG(objects_store).top++;
    }
    object->handle = handle;
    //存入object_buckets数组
    EG(objects_store).object_buckets[handle] = object;
}

typedef struct _zend_objects_store {
    zend_object **object_buckets; //对象数组
    uint32_t top; //当前全部object数
    uint32_t size; //object_buckets大小
    int free_list_head; //第一个可用object_buckets位置
} zend_objects_store;
```

将所有的对象保存在 `EG(objects_store).object_buckets` 中的目的是用于垃圾回收(不确定是不是还有其它的作用)，防止出现循环引用而导致内存泄漏的问题，这个机制后面章节会单独介绍，这里只要记得有这么个东西就行了。

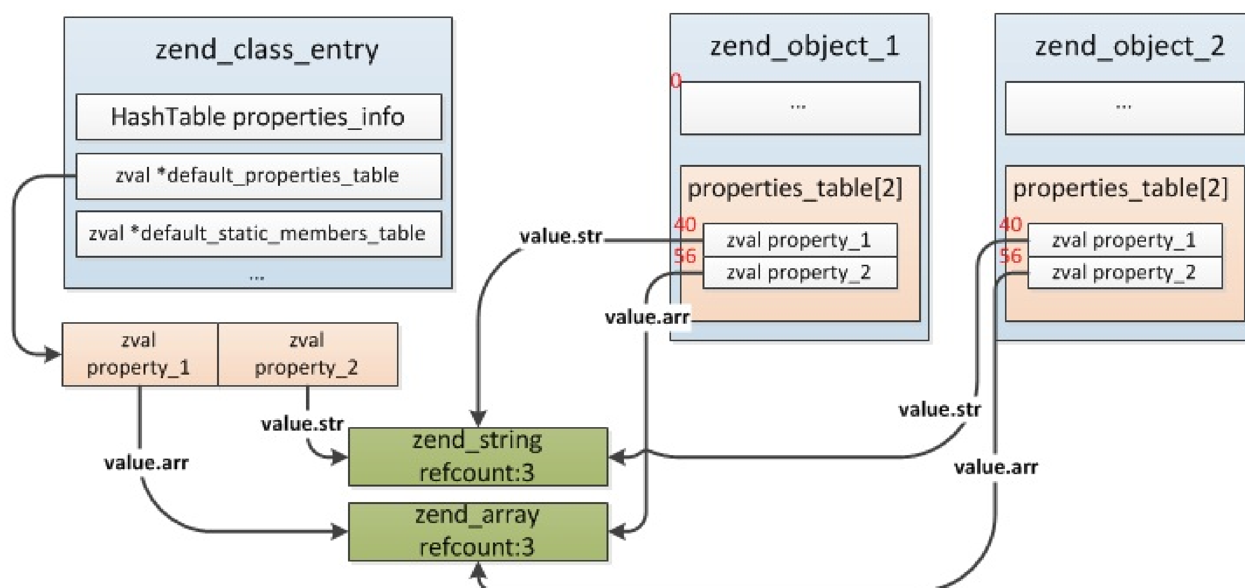
(2)初始化成员属性

```
ZEND_API void object_properties_init(zend_object *object, zend_c
lass_entry *class_type)
{
    if (class_type->default_properties_count) {
        zval *src = class_type->default_properties_table;
        zval *dst = object->properties_table;
        zval *end = src + class_type->default_properties_count;

        //将非静态属性值从：
        //zend_class_entry.default_properties_table复制到zend_obj
ect.properties_table
        do {
            ZVAL_COPY(dst, src);
            src++;
            dst++;
        } while (src != end);
        object->properties = NULL;
    }
}
```

这一步操作是将非静态属性的值从 `zend_class_entry.default_properties_table -> zend_object.properties_table` ，当然这里不是硬拷贝，而是浅复制(增加引用)，两者当前指向的value还是同一份，除非对象试图改写指向的属性值，那时将触发写时复制机制重新拷贝一份。

上面那个例子，类有两个普通属性：$name、$ids，假如我们实例化了两个对象，那么zend_class_entry与zend_object中普通属性值的关系如下图所示。



以上就是实例化一个对象的过程，总结一下具体的步骤：

- **step1:** 首先根据类名去EG(class_table)中找到具体的类，即zend_class_entry
- **step2:** 分配zend_object结构，一起分配的还有普通非静态属性值的内存
- **step3:** 初始化对象的非静态属性，将属性值从zend_class_entry浅复制到对象中
- **step4:** 查找当前类是否定义了构造函数，如果没有定义则跳过执行构造函数的opcode，否则为调用构造函数的执行进行一些准备工作(分配zend_execute_data)
- **step5:** 实例化完成，返回新实例化的对象(如果返回的对象没有变量使用则直接释放掉了)

3.4.2.3 成员属性的读写

普通成员属性的读写处理handler分别为 `zend_object.handlers` 中的：read_property、write_property，默认对应的函数为：zend_std_read_property()、zend_std_write_property()，访问获取修改一个普通成员属性时就是由这两个函数完成的。

(1)读取属性：

通过对象或方法内通过$this访问属性，比如： `echo $obj->name;` ，具体的实现：

```
zval *zend_std_read_property(zval *object, zval *member, int typ
e, void **cache_slot, zval *rv)
```

```
{
    zend_object *zobj;
    uint32_t property_offset;

    zobj = Z_OBJ_P(object);

    //根据属性名在zend_class.zend_property_info中查找zend_property_
info，得到属性值在zend_object中的存储offset
    //注意：zend_get_property_offset()会对属性的可见性(public、priva
te、protected)进行验证
    property_offset = zend_get_property_offset(zobj->ce, Z_STR_P
(member), (type == BP_VAR_IS) || (zobj->ce->__get != NULL), cach
e_slot);

    if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET))
 {
        if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OF
FSET)) {
            //普通属性，直接根据offset取到属性值：((zval*)((char*)(zo
bj) + offset))
            retval = OBJ_PROP(zobj, property_offset);
        } else if (EXPECTED(zobj->properties != NULL)) {
            //动态属性的情况，没有在类中显式定义的属性，后面一节会单独介绍
            ....
        }
    } else if (UNEXPECTED(EG(exception))) {
        ...
    }

    //没有找到属性
    //调用魔术方法：__isset()
    if ((type == BP_VAR_IS) && zobj->ce->__isset) {
        ...
    }

    //调用魔术方法：__get()
    if (zobj->ce->__get) {
        zend_long *guard = zend_get_property_guard(zobj, Z_STR_P
(member));
        ...
```

```
        if(!((*guard) & IN_ISSET)){
            *guard |= IN_ISSET;
            zend_std_call_issetter(&tmp_object, member, &tmp_res
ult);
            *guard &= ~IN_ISSET;
            ...
        }
    }
    ...
}
```

普通成员属性的查找比较容易理解，首先是从zend_class的属性信息哈希表中找到zend_property_info，并判断其可见性(public、private、protected)，如果可以访问则直接根据属性的offset在zend_object.properties_table数组中取到属性值，如果没有在属性哈希表中找到且定义了**get()**魔术方法则会调用get()方法处理。

> **Note:** 如果类存在**get()**方法，则在实例化对象分配属性内存**(即:properties_table)**时会多分配一个**zval**，类型为**HashTable**，每次调用get($var)时会把输入的$var名称存入这个哈希表，这样做的目的是防止循环调用，举个例子：
>
> ***public function __get($var) { return $this->$var; }***
>
> 这种情况是调用**get()**时又访问了一个不存在的属性，也就是会在get()方法中递归调用，如果不对请求的$var作判断则将一直递归下去，所以在调用**get()**前首先会判断当前**$var**是不是已经在get()中了，如果是则不会再调用**get()**，否则会把**$var**作为**key**插入那个**HashTable**，然后将哈希值设置为：***guard |= IN_ISSET**，调用完get()再把哈希值设置为：*guard &= ~IN_ISSET。
>
> 这个HashTable不仅仅是给**get()**用的，其它魔术方法也会用到，所以其哈希值类型是**zend_long**，不同的魔术方法占不同的**bit**位；其次，并不是所有的对象都会额外分配这个**HashTable**，在对象创建时会根据 ***zend_class_entry.ce_flags*** 是否包含 ***ZEND_ACC_USE_GUARDS*** 确定是否分配，在类编译时如果发现定义了get()、**set()**、unset()、__isset()方法则会将ce_flags打上这个掩码。

(2)设置属性：

与读取属性不同，设置属性是对属性的修改操作，比如： `$obj->name = "pangudashu";` ，看下具体的实现过程：

```
ZEND_API void zend_std_write_property(zval *object, zval *member
, zval *value, void **cache_slot)
{
    zend_object *zobj;
    uint32_t property_offset;

    zobj = Z_OBJ_P(object);

    //与读取属性相同
    property_offset = zend_get_property_offset(zobj->ce, Z_STR_P
(member), (zobj->ce->__set != NULL), cache_slot);

    if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET))
 {
        if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OF
FSET)) {
            //普通属性
            variable_ptr = OBJ_PROP(zobj, property_offset);
            if (Z_TYPE_P(variable_ptr) != IS_UNDEF) {
                goto found;
            }
        } else if (EXPECTED(zobj->properties != NULL)) {
            //动态属性哈希表已经初始化，直接插入zobj->properties哈希表,
后面单独介绍
            ...
            if ((variable_ptr = zend_hash_find(zobj->properties,
 Z_STR_P(member))) != NULL) {
found:
                //赋值操作，与普通变量的操作相同
                zend_assign_to_variable(variable_ptr, value, IS_
CV);
                goto exit;
            }
        }
    } else if (UNEXPECTED(EG(exception))) {
        ...
    }
```

```
    //没有找到属性
    //如果定义了__set()则调用
    if (zobj->ce->__set) {
        //与__get()相同，也会判断set的变量名是否已经在__set()中
        ...
        ZVAL_COPY(&tmp_object, object);
        (*guard) |= IN_SET; //防止循环__set()
        if (zend_std_call_setter(&tmp_object, member, value) !=
SUCCESS) {
        }
        (*guard) &= ~IN_SET;
    }else if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OF
FSET)) {
        ...
    }
}
```

首先与读取属性的操作相同：先找到zend_property_info，判断其可见性，然后根据offset取到具体的属性值，最后对其进行赋值修改。

> **Note:** 属性读写操作的函数中有一个cache_slot的参数，它的作用涉及PHP的一个缓存机制：运行时缓存，后面会单独介绍。

3.4.2.4 对象的复制

PHP中普通变量的复制可以通过直接赋值完成，比如：

```
$a = array();
$b = $a;
```

但是对象无法这么进行复制，仅仅通过赋值传递对象，它们指向的都是同一个对象，修改时也不会发生硬拷贝。比如上面这个例子，我们把 `$a` 赋值给 `$b` ，然后如果我们修改 `$b` 的内容，那么这时候会进行value分离， `$a` 的内容是不变的，但是如果是把一个对象赋值给了另一个变量，这俩对象不管哪一个修改另外一个都随之改变。

```
class my_class
{
    public $arr = array();
}

$a = new my_class;
$b = $a;

$b->arr[] = 1;

var_dump($a === $b);
====================
输出：bool(true)
```

还记得我们在《2.1.3.2 写时复制》一节讲过zval有个类型掩码: **type_flag** 吗？其中有个是否可复制的标识：**IS_TYPE_COPYABLE** ，copyable的意思是当value发生duplication时是否需要或能够copy，而object的类型是不能复制(不清楚的可以翻下前面的章节)，所以我们不能简单的通过赋值语句进行对象的复制。

PHP提供了另外一个关键词来实现对象的复制：**clone**。

```
$copy_of_object = clone $object;
```

`clone` 出的对象就与原来的对象完全隔离了，各自修改都不会相互影响，另外如果类中定义了 `__clone()` 魔术方法，那么在 `clone` 时将调用此函数。

`clone` 的实现比较简单，通过 `zend_object.clone_obj` (即: `zend_objects_clone_obj()`)完成。

```
//zend_objects.c
ZEND_API zend_object *zend_objects_clone_obj(zval *zobject)
{
    zend_object *old_object;
    zend_object *new_object;

    old_object = Z_OBJ_P(zobject);
    //重新分配一个zend_object
    new_object = zend_objects_new(old_object->ce);

    //浅复制properties_table、properties
    //如果定义了__clone()则调用此方法
    zend_objects_clone_members(new_object, old_object);

    return new_object;
}
```

3.4.2.5 对象比较

当使用比较运算符（==）比较两个对象变量时，比较的原则是：如果两个对象的属性和属性值 都相等，而且两个对象是同一个类的实例，那么这两个对象变量相等；而如果使用全等运算符（===），这两个对象变量一定要指向某个类的同一个实例（即同一个对象）。

PHP中对象间的"=="比较通过函数 `zend_std_compare_objects()` 处理。

```
static int zend_std_compare_objects(zval *o1, zval *o2)
{
    ...

    if (zobj1->ce != zobj2->ce) {
        return 1; /* different classes */
    }
    if (!zobj1->properties && !zobj2->properties) {
        //逐个比较properties_table
        ...
    }else{
        //比较properties
        return zend_compare_symbol_tables(zobj1->properties, zob
j2->properties);
    }
}
```

"==="的比较通过函数`zend_is_identical()`处理，比较简单，这里不再展开。

3.4.2.6 对象的销毁

object与string、array等类型不同，它是个复合类型，所以它的销毁过程更加复杂，赋值、函数调用结束或主动unset等操作中如果发现object引用计数为0则将触发销毁动作。

```
//情况1
$obj1 = new my_function();

$obj1 = 123; //此时将断开对zend_object的引用，如果refcount=0则销毁

//情况2
function xxxx(){
    $obj1 = new my_function();
    ...
    return null; //清理局部变量时如果发现$obj1引用为0则销毁
}

//情况3
$obj1 = new my_function();
//整个脚本结束，清理全局变量时

//情况4
$obj1 = new my_function();
unset($obj1);
```

上面这几个都是比较常见的会进行变量销毁的情况，销毁一个对象由 `zend_objects_store_del()` 完成，销毁的过程主要是清理成员属性、从EG(objects_store).object_buckets中删除、释放zend_object内存等等。

```
//zend_objects_API.c
ZEND_API void zend_objects_store_del(zend_object *object)
{
    //这个函数if嵌套写的很挫...
    ...
    if (GC_REFCOUNT(object) > 0) {
        GC_REFCOUNT(object)--;
        return;
    }
    ...

    //调用dtor_obj，默认zend_objects_destroy_object()
    //接着调用free_obj，默认zend_object_std_dtor()
    object->handlers->dtor_obj(object);
    object->handlers->free_obj(object);
    ...
    ptr = ((char*)object) - object->handlers->offset;
    efree(ptr);
}
```

另外，在减少refcount时如果发现object的引用计数大于0那么并不是什么都不做了，还记得2.1.3.4介绍的垃圾回收吗？PHP变量类型有的会因为循环引用导致正常的gc无法生效，这种类型的变量就有可能成为垃圾，所以会对这些类型的 `zval.u1.type_flag` 打上 `IS_TYPE_COLLECTABLE` 标签，然后在减少引用时即使refcount大于0也会启动垃圾检查，目前只有object、array两种类型会使用这种机制。

3.4.3 继承

继承是面向对象编程技术的一块基石，它允许创建分等级层次的类，它允许子类继承父类所有公有或受保护的特征和行为，使得子类对象具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

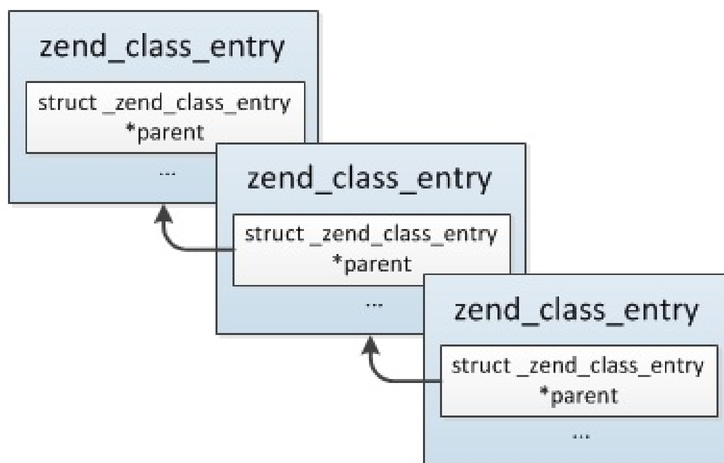
继承对于功能的设计和抽象是非常有用的，而且对于类似的对象增加新功能就无须重新再写这些公用的功能。

PHP中通过 `extends` 关键词继承一个父类，一个类只允许继承一个父类，但是可以多级继承。

```
class 父类 {
}

class 子类 extends 父类 {
}
```

前面的介绍我们已经知道，类中保存着成员属性、方法、常量等，父类与子类之间通过 `zend_class_entry.parent` 建立关联，如下图所示。



问题来了：每个类都有自己独立的常量、成员属性、成员方法，那么继承类父子之间的这些信息是如何进行关联的呢？接下来我们将带着这个疑问再重新分析一下类的编译过程中是如何处理继承关系的。

3.4.1.5一节详细介绍了类的编译过程，这里再简单回顾下：首先为类分配一个zend*classentry*结构，如果没有继承类则生成一条类声明的*opcode(ZENDDECLARECLASS)*，有继承类则生成两条*opcode(ZENDFETCHCLASS*、*ZENDDECLAREINHERITED_CLASS)*，然后再继续编译常量、成员属性、成员方法注册到*zend_class_entry*中，最后编译完成后调用 `zend_do_early_binding()` 进行 父子类关联 以及 注册到*EG(class_table)*符号表。

如果父类在子类之前定义的，那么父子类之间的关联就是在 `zend_do_early_binding()` 中完成的，这里不考虑子类在父类前定义的情况，实际两者没有本质差别，区别在于在哪一个阶段执行。有继承类的情况在 `zend_do_early_binding()` 中首先是查找父类，然后调用 `do_bind_inherited_class()` 处理，最后将 `ZEND_FETCH_CLASS` 、 `ZEND_DECLARE_INHERITED_CLASS` 两条opcode删除，这些过程前面已经介绍过了，下面我们重点看下 `do_bind_inherited_class()` 的处理过程。

```
ZEND_API zend_class_entry *do_bind_inherited_class(
    const zend_op_array *op_array, //这个是定义类的地方的
    const zend_op *opline, //类声明的opcode：ZEND_DECLARE_INHERITE
D_CLASS
    HashTable *class_table, //CG(class_table)
    zend_class_entry *parent_ce,  //父类
    zend_bool compile_time) //是否编译时
{
     zend_class_entry *ce;
     zval *op1, *op2;

     if (compile_time) {
        op1 = CT_CONSTANT_EX(op_array, opline->op1.constant);
        op2 = CT_CONSTANT_EX(op_array, opline->op2.constant);
     }else{
        ...
     }
     ...
     //父子类关联
     zend_do_inheritance(ce, parent_ce);

     //注册到CG(class_table)
     ...
}
```

上面这个函数的处理与注册非继承类的 `do_bind_class()` 几乎完全相同，只是多了一个 `zend_do_inheritance()` 一步，此函数输入很直观，只一个类及父类。

```
//zend_inheritance.c #line:758
ZEND_API void zend_do_inheritance(zend_class_entry *ce, zend_class_entry *parent_ce)
{
    zend_property_info *property_info;
    zend_function *func;
    zend_string *key;
    zval *zv;

    //interface、trait、final类检查
    ...
    ce->parent = parent_ce;

    zend_do_inherit_interfaces(ce, parent_ce);

    //下面就是继承属性、常量、方法
}
```

下面的操作我们根据一个示例逐个来看。

```
//示例
class A {
    const A1 = 1;
    public $a1 = array(1);
    private $a2 = 120;

    public function get() {
        echo "A::get()";
    }
}
class B extends A {
    const B1 = 2;

    public $b1 = "ddd";

    public function get() {
        echo "B::get()";
    }
}
```

3.4.3.1 继承属性

前面我们已经介绍过：属性按静态、非静态分别保存在两个数组中，各属性按照定义的先后顺序编号(offset)，同时按照这个编号顺序存储排列，而这些编号信息通过 `zend_property_info` 结构保存，全部静态、非静态属性的 `zend_property_info` 保存在一个以属性名为key的HashTable中，所以检索属性时首先根据属性名找到此属性的 `zend_property_info` ，然后拿到其属性值的offset，再根据静态、非静态分别到 `default_static_members_count` 、 `default_properties_table` 数组中取出属性值。

当类存在继承关系时，操作方式是：将属性从父类复制到子类 。子类会将父类的公共、受保护的属性值数组全部合并到子类中，然后将全部属性的 `zend_property_info` 哈希表也合并到子类中。

合并的步骤:

(1)合并非静态属性**default_properties_table:** 首先申请一个父类+子类非静态属性大小的数组，然后先将父类非静态属性复制到新数组，然后再将子类的非静态数组接着父类属性的位置复制过去，子类的default_properties_table指向合并后的新数组，default_properties_count更新为新数组的大小，最后将子类旧的数组释放。

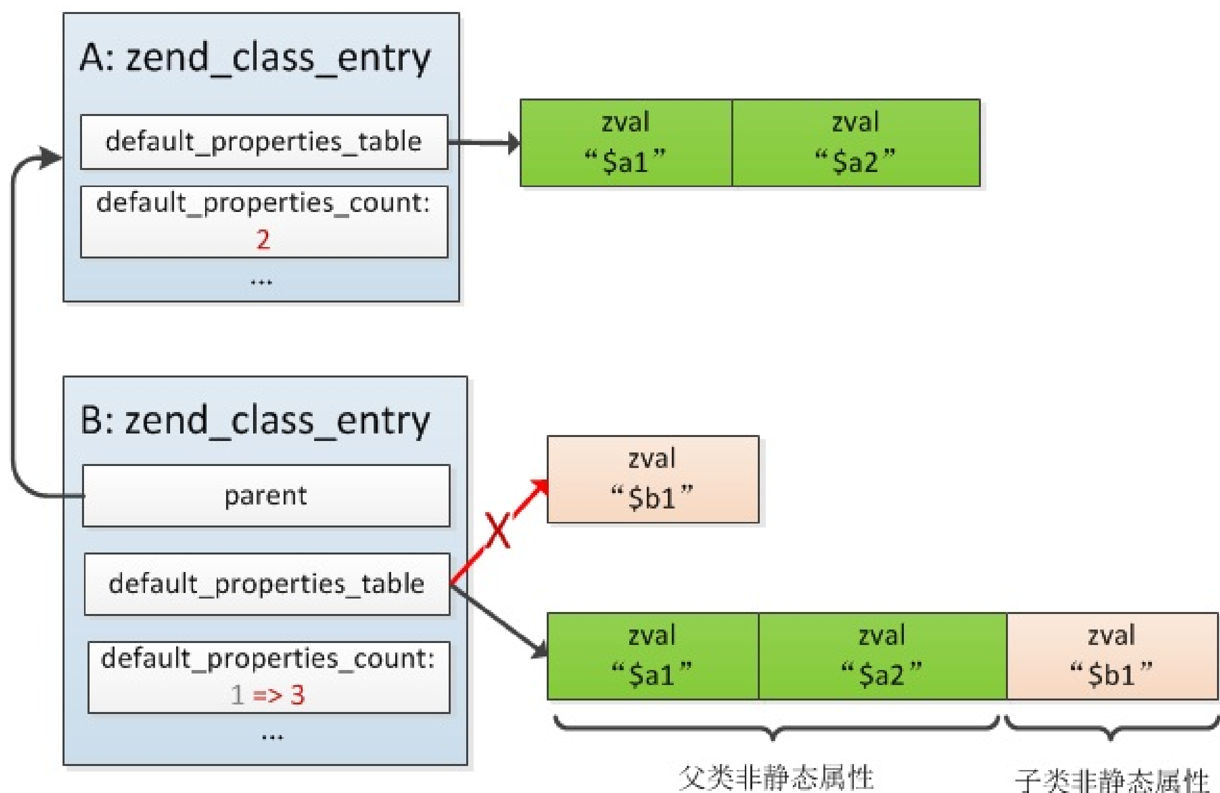
```
if (parent_ce->default_properties_count) {
    zval *src, *dst, *end;
    ...
    zval *table = pemalloc(sizeof(zval) * (ce->default_propertie
s_count + parent_ce->default_properties_count), ...);

    ce->default_properties_table = table;

    //复制父类、子类default_properties_table
    do {
        ...
    }while(dst != end);

    //更新default_properties_count为合并后的大小
    ce->default_properties_count += parent_ce->default_propertie
s_count;
}
```

示例合并后的情况如下图。



(2)合并静态属性**default_static_members_table:** 与非静态属性相同，新申请一个父类+子类静态属性大小的数组，依次将父类、子类静态属性复制到新数组，然后更新子类default_static_members_table指向新数组。

(3)更新子类属性**offset:** 因为合并后原子类属性整体向后移了，所以子类属性的编号offset需要加上前面父类属性的总大小。

```
ZEND_HASH_FOREACH_PTR(&ce->properties_info, property_info) {
    if (property_info->ce == ce) {
        if (property_info->flags & ZEND_ACC_STATIC) {
            //静态属性offset为数组下标，直接加上父类default_static_members_count即可
            property_info->offset += parent_ce->default_static_members_count;
        } else {
            //非静态属性offset为内存偏移值，按zval大小递增
            property_info->offset += parent_ce->default_properties_count * sizeof(zval);
        }
    }
} ZEND_HASH_FOREACH_END();
```

(4)合并**properties_info**哈希表**:** 这也是非常关键的一步，上面只是将父类的属性值合并到了子类，但是索引属性用的是properties_info哈希表，所以需要将父类的属性索引表与子类的索引表合并。在合并的过程中就牵扯到父子类属性的继承、覆盖问题了，各种情况具体处理如下：

- 父类属性不与子类冲突 且 父类属性是私有**:** 即父类属性为private，且子类中没有重名的，则将此属性插入子类properties_info，但是更新其flag为ZEND_ACC_SHADOW，这种属性将不能被子类使用；
- 父类属性不与子类冲突 且 父类属性是公有**:** 这种比较简单，子类可以继承使用，直接插入子类properties_info；
- 父类属性与子类冲突 且 父类属性为私有**:** 不继承父类的，以子类原属性为准，但是打上 `ZEND_ACC_CHANGED` 的flag，这种属性父子类隔离，互不干扰；
- 父类属性与子类冲突 且 父类属性是公有或受保护的**:**
    - 父子类属性一个是静态一个是非静态**:** 编译错误；
    - 父子类属性都是非静态**:** 用父类的offset，但是值用子类的，父子类共享；
    - 父子类属性都是静态**:** 不继承父类属性，以子类原属性为准，父子类隔离，互不干扰；

这个地方相对比较复杂，具体的合并策略在 `do_inherit_property()` 中，这里不再罗列代码。

所以，继承类实际上是把父类的属性、常量、方法合并到了子类里面，上一节介绍实例化时会将普通成员属性值复制到对象中去，这样在实例化时子类就与普通的类的操作没有任何差别了。

3.4.3.2 继承常量

常量的合并策略比较简单，如果父类与子类冲突时用子类的，不冲突时则将父类的常量合并到子类。

```
static void do_inherit_class_constant(zend_string *name, zval *z
v, zend_class_entry *ce, zend_class_entry *parent_ce)
{
    //父类定义的常量在子类中没有定义
    if (!zend_hash_exists(&ce->constants_table, name)) {
        ...
        _zend_hash_append(&ce->constants_table, name, zv);
    }
}
```

3.4.3.3 继承方法

与属性一样，子类可以继承父类的公有、受保护的方法，方法的继承比较复杂，因为会有访问控制、抽象类、接口、Trait等多种限制条件。实现上与前面几种相同，即父类的function_table合并到子类的function_table中。

首先是将子类function_table扩大，以容纳父子类全部方法，然后遍历父类function_table，逐个判断是否可被子类继承，如果可被继承则插入到子类function_table中。

```
if (zend_hash_num_elements(&parent_ce->function_table)) {
    //扩展子类的function_table哈希表大小
    zend_hash_extend(&ce->function_table,
            zend_hash_num_elements(&ce->function_table) +
            zend_hash_num_elements(&parent_ce->function_table), 0
);

    //遍历父类function_table，检查是否可被子类继承
    ZEND_HASH_FOREACH_STR_KEY_PTR(&parent_ce->function_table, ke
y, func) {
        zend_function *new_func = do_inherit_method(key, func, c
e);

        if (new_func) {
            _zend_hash_append_ptr(&ce->function_table, key, new_
func);
        }
    } ZEND_HASH_FOREACH_END();
}
```

在合并的过程中需要对父类的方法进行一系列检查，最简单的情况就是父类中定义的方法在子类中不存在，这种情况比较简单，直接将父类的zend_function复制一份给子类。

```
static zend_function *do_inherit_method(zend_string *key, zend_f
unction *parent, zend_class_entry *ce)
{
    zval *child = zend_hash_find(&ce->function_table, key);

    if(child){
        //方法与子类冲突
        ...
    }

    //父子类方法不冲突，直接复制
    return zend_duplicate_function(parent, ce);
}
```

当然这里不完全是复制：如果继承的父类是内部类则会硬拷贝一份zend_function结构(此结构的指针成员不复制)；如果父类是用户自定义的类，且继承的方法没有静态变量则不会硬拷贝，而是增加zend_function的引用计数(zend_op_array.refcount)。

```
//func是父类成员方法，ce是子类
static zend_function *zend_duplicate_function(zend_function *func, zend_class_entry *ce)
{
    zend_function *new_function;

    if (UNEXPECTED(func->type == ZEND_INTERNAL_FUNCTION)) {
        //内部函数
        //如果子类也是内部类则会调用malloc分配内存(不会被回收)，否则在zend内存池分配
        ...
    }else{
        if (func->op_array.refcount) {
            (*func->op_array.refcount)++;
        }
        if (EXPECTED(!func->op_array.static_variables)) {
            return func;
        }

        //硬拷贝
        new_function = zend_arena_alloc(&CG(arena), sizeof(zend_op_array));
        memcpy(new_function, func, sizeof(zend_op_array));
    }
}
```

合并时另外一个比较复杂的情况是父类与子类中的方法冲突了，即子类重写了父类的方法，这种情况需要对父子类以及要合并的方法进行一系列检查，这一步在 `do_inheritance_check_on_method()` 中完成，具体情况如下：

```
static void do_inheritance_check_on_method(zend_function *child,
 zend_function *parent)
{
    uint32_t child_flags;
    uint32_t parent_flags = parent->common.fn_flags;
    ...
}
```

(1)抽象子类的抽象方法与抽象父类的抽象方法冲突**:** 无法重写，Fatal错误。

```
abstract class B extends A {
    abstract function test();
}
abstract class A
{
    abstract function test();
}

============================
PHP Fatal error:  Can't inherit abstract function A::test() (pre
viously declared abstract in B)
```

判断逻辑：

```
//do_inheritance_check_on_method():

if ((parent->common.scope->ce_flags & ZEND_ACC_INTERFACE) == 0 //父类非接口
        && parent->common.fn_flags & ZEND_ACC_ABSTRACT //父类方法为抽象方法
        && parent->common.scope != (child->common.prototype ? child->common.prototype->common.scope : child->common.scope)
        && child->common.fn_flags & (ZEND_ACC_ABSTRACT|ZEND_ACC_IMPLEMENTED_ABSTRACT) //子类方法为抽象或实现了抽象方法
) {
    zend_error_noreturn(E_COMPILE_ERROR, "Can't inherit abstract function %s::%s() (previously declared abstract in %s)",...);
}
```

(2)父类方法为**final:** Fatal错误，final成员方法不得被重写。 判断逻辑：

```
//do_inheritance_check_on_method():

if (UNEXPECTED(parent_flags & ZEND_ACC_FINAL)) {
    zend_error_noreturn(E_COMPILE_ERROR, "Cannot override final method %s::%s()", ...);
}
```

(3)父子类方法静态属性不一致**:** 父类方法为非静态而子类的是静态(或相反)，Fatal错误。

```
class A {
    public function test(){}
}

class B extends A {
    static public function test(){}
}

============================
PHP Fatal error:  Cannot make non static method A::test() static
 in class B
```

判断逻辑：

```
//do_inheritance_check_on_method():

if (UNEXPECTED((child_flags & ZEND_ACC_STATIC) != (parent_flags
& ZEND_ACC_STATIC))) {
    zend_error_noreturn(E_COMPILE_ERROR,...);
}
```

(4)抽象子类的抽象方法覆盖父类非抽象方法: Fatal错误。

```
class A {
    public function test(){}
}

abstract class B extends A {
    abstract public function test();
}

============================
PHP Fatal error:  Cannot make non abstract method A::test() abst
ract in class B
```

判断逻辑：

```
//do_inheritance_check_on_method():

if (UNEXPECTED((child_flags & ZEND_ACC_ABSTRACT) > (parent_flags
 & ZEND_ACC_ABSTRACT))) {
    zend_error_noreturn(E_COMPILE_ERROR, "Cannot make non abstra
ct method %s::%s() abstract in class %s",...);
}
```

(5)子类方法限制父类方法访问权限: Fatal错误，不允许派生类限制父类方法的访问权限，如父类方法为public，而子类试图重写为protected/private。

```
class A {
    public function test(){}
}

class B extends A {
    protected function test(){}
}

============================
PHP Fatal error:  Access level to B::test() must be public (as i
n class A)
```

判断逻辑：

```
//do_inheritance_check_on_method():

//ZEND_ACC_PPP_MASK = (ZEND_ACC_PUBLIC | ZEND_ACC_PROTECTED | ZE
ND_ACC_PRIVATE)
if (UNEXPECTED((child_flags & ZEND_ACC_PPP_MASK) > (parent_flags
 & ZEND_ACC_PPP_MASK))) {
    zend_error_noreturn(E_COMPILE_ERROR, "Access level to %s::%s
() must be %s (as in class %s)%s", ...);
} else if (((child_flags & ZEND_ACC_PPP_MASK) < (parent_flags &
ZEND_ACC_PPP_MASK))
        && ((parent_flags & ZEND_ACC_PPP_MASK) & ZEND_ACC_PRIVAT
E)) {
    child->common.fn_flags |= ZEND_ACC_CHANGED;
}
```

(6)剩余检查情况: 除了上面5中情形下无法重写方法，剩下还有一步对函数参数的检查，这个过程我们整体看一下。

```
//do_inheritance_check_on_method():

if (UNEXPECTED(!zend_do_perform_implementation_check(child, pare
nt))) {
    ...
    zend_error(error_level, "Declaration of %s %s be compatible
with %s", ZSTR_VAL(child_prototype), error_verb, ZSTR_VAL(method
_prototype));
    zend_string_free(child_prototype);
    zend_string_free(method_prototype);
}
```

实际上 `zend_do_perform_implementation_check()` 这个函数是用来检查一个方法是否实现了某抽象方法的，继承的时候遵循的也是这个规则，所以这里可以将父类方法理解为抽象方法，只有子类方法实现了该"抽象方法"才能重写父类方法。

```
static zend_bool zend_do_perform_implementation_check(const zend
_function *fe, const zend_function *proto)
{
    ...
```

```
    //如果检查的方法是__construct且父类方法不是interface和abstract则子类__construct覆盖父类的
    if ((fe->common.fn_flags & ZEND_ACC_CTOR)
        && ((proto->common.scope->ce_flags & ZEND_ACC_INTERFACE) == 0
            && (proto->common.fn_flags & ZEND_ACC_ABSTRACT) == 0)) {
        return 1;
    }

    //如果父类方法为私有方法则子类方法可以覆盖
    if (proto->common.fn_flags & ZEND_ACC_PRIVATE) {
        return 1;
    }

    //如果父类方法必传参数小于子类的或者父类的总参数大于子类的则不能覆盖
    //如：
    //  父类 public function test($a, $b = 3){}
    //  子类 public function test($a, $b){}
    if (proto->common.required_num_args < fe->common.required_num_args
        || proto->common.num_args > fe->common.num_args) {
        return 0;
    }

    //可变函数,暂未理解这里的可变函数指哪类，忽略
    ...

    //如果有定义的参数检查参数类型是否匹配，如果显式声明了参数类型则父子类方法必须匹配
    for (i = 0; i < num_args; i++) {
        zend_arg_info *fe_arg_info = &fe->common.arg_info[i];
        if (!zend_do_perform_type_hint_check(fe, fe_arg_info, proto, proto_arg_info)) {
            return 0;
        }

        //是否引用也必须一致
        if (fe_arg_info->pass_by_reference != proto_arg_info->pass_by_reference) {
```

```
            return 0;
        }
    }

    //如果父类方法声明了返回值类型则子类方法必须声明且类型一致，相反如果子类声明了而父类无要求则可以
    if (proto->common.fn_flags & ZEND_ACC_HAS_RETURN_TYPE) {
        if (!(fe->common.fn_flags & ZEND_ACC_HAS_RETURN_TYPE)) {
            return 0;
        }

        if (!zend_do_perform_type_hint_check(fe, fe->common.arg_info - 1, proto, proto->common.arg_info - 1)) {
            return 0;
        }
    }
}
```

这个判断过程还是比较复杂的，有些地方很难理解为什么设计，想了解完整过程的可以自行翻下代码。

3.4.4 动态属性

前面介绍的成员属性都是在类中明确的定义过的，这些属性在实例化时会被拷贝到对象空间中去，PHP中除了显示的在类中定义成员属性外，还可以动态的创建非静态成员属性，这种属性不需要在类中明确定义，可以直接通过： `$obj->property_name=xxx` 、 `$this->property_name = xxx` 为对象设置一个属性，这种属性称之为动态属性，举个例子：

```
class my_class {
    public $id = 123;

    public function test($name, $value){
        $this->$name = $value;
    }
}

$obj = new my_class;
$obj->test("prop_1", array(1,2,3));
//或者直接：
//$obj->prop_1 = array(1,2,3);

print_r($obj);
```

在 `test()` 方法中直接操作了没有定义的成员属性，上面的例子将输出：

```
my_class Object
(
    [id] => 123
    [prop_1] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
        )
)
```

前面类、对象两节曾介绍，非静态成员属性值在实例化时保存到了对象中，属性的操作按照编译时按顺序编好的序号操作，各对象对其非静态成员属性的操作互不干扰，那么动态属性是在运行时创建的，它是如何存储的呢？

与普通非静态属性不同，动态创建的属性保存在 `zend_object->properties` 哈希表中，查找的时候首先按照普通属性在 `zend_class_entry.properties_info` 找，没有找到再去 `zend_object->properties` 继续查找。动态属性的创建过程(即：修改属性的操作)：

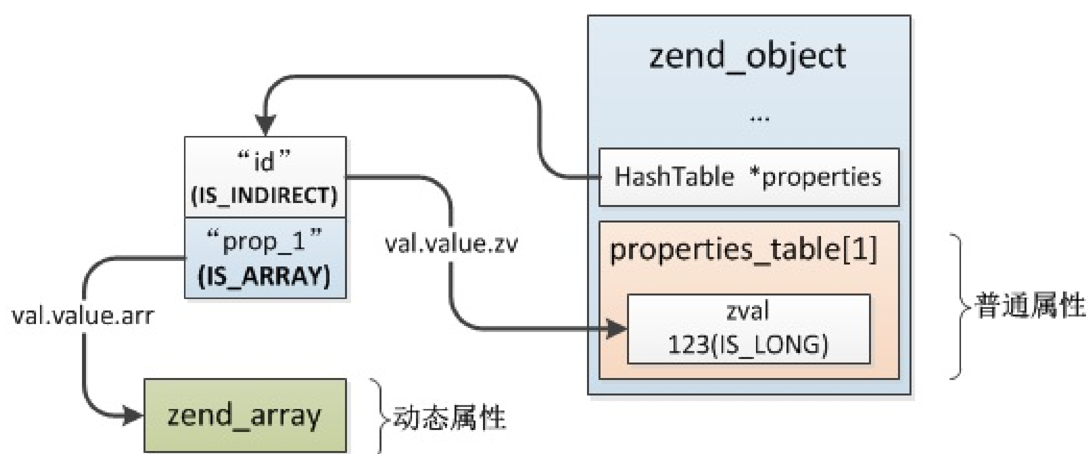
```
//zend_object->handlers->write_property:
ZEND_API void zend_std_write_property(zval *object, zval *member
, zval *value, void **cache_slot)
{
    ...
    zobj = Z_OBJ_P(object);
    //先在zend_class_entry.properties_info查找此属性
    property_offset = zend_get_property_offset(zobj->ce, Z_STR_P
(member), (zobj->ce->__set != NULL), cache_slot);

    if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET))
 {
        if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OF
FSET)) {
            //普通属性，直接根据根据属性ofsset取出属性值
        } else if (EXPECTED(zobj->properties != NULL)) { //有动态
属性
            ...
            //从动态属性中查找
            if ((variable_ptr = zend_hash_find(zobj->properties,
 Z_STR_P(member))) != NULL) {
found:
                zend_assign_to_variable(variable_ptr, value, IS_
CV);
                goto exit;
            }
        }
    }

    if (zobj->ce->__set) {
```

```
        //定义了__set()魔法函数
    }else if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OF
FSET)){
        if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OF
FSET)) {
            ...
        } else {
            //首次创建动态属性将在这里完成
            if (!zobj->properties) {
                rebuild_object_properties(zobj);
            }
            //将动态属性插入properties
            zend_hash_add_new(zobj->properties, Z_STR_P(member),
 value);
        }
    }
}
```

上面就是成员属性的修改过程，普通属性根据其offset再从对象中取出属性值进行修改，而首次创建动态属性将通过 `rebuild_object_properties()` 初始化 `zend_object->properties` 哈希表，后面再创建动态属性直接插入此哈希表， `rebuild_object_properties()` 过程并不仅仅是创建一个HashTable，还会将普通成员属性值插入到这个数组中，与动态属性不同，这里的插入并不是增加原zend_value的refcount，而是创建了一个IS_INDIRECT类型的zval，指向原属性值zval，具体结构如下图。



> **Note:** 这里不清楚将原有属性也插入properties的用意，已知用到的一个地方是在GC垃圾回收获取对象所有属性时(zend_std_get_gc())，如果有动态属性则直接返回properties给GC遍历，假如不把普通的显式定义的属性"拷贝"进来则需要返回、遍历两个数组。
>
> 另外一个地方需要注意，把原属性"转移"到properties并不仅仅是创建动态属性时触发的，调用对象的get_properties(即：zend_std_get_properties())也会这么处理，比如将一个object转为array时就会触发这个动作: $arr = (array)$object，通过foreach遍历一个对象时也会调用get_properties获取属性数组进行遍历。

成员属性的读取通过 `zend_object->handlers->read_property` (默认zend_std_read_property())函数完成，动态属性的查找过程实际与 `write_property` 中相同：

```
zval *zend_std_read_property(zval *object, zval *member, int typ
e, void **cache_slot, zval *rv)
{
    ...
    zobj = Z_OBJ_P(object);

    //首先查找zend_class_entry.properties_info，普通属性可以在这里找到

    property_offset = zend_get_property_offset(zobj->ce, Z_STR_P
(member), (type == BP_VAR_IS) || (zobj->ce->__get != NULL), cach
e_slot);

    if (EXPECTED(property_offset != ZEND_WRONG_PROPERTY_OFFSET))
 {
        if (EXPECTED(property_offset != ZEND_DYNAMIC_PROPERTY_OF
FSET)) {
            //普通属性
            retval = OBJ_PROP(zobj, property_offset);
        } else if (EXPECTED(zobj->properties != NULL)) {
            //动态属性从zend_object->properties中查找
            retval = zend_hash_find(zobj->properties, Z_STR_P(me
mber));
            if (EXPECTED(retval)) goto exit;
        }
    }
    ...
}
```

3.4.5 魔术方法

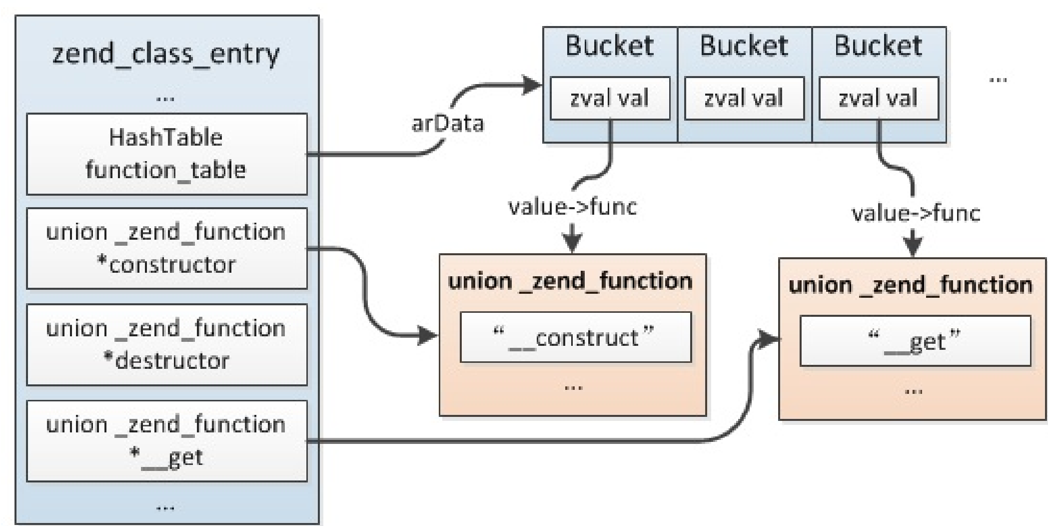
PHP在类的成员方法中预留了一些特殊的方法，它们会在一些特殊的时机被调用(比如创建对象之初、访问成员属性时...)，这类方法称为：魔术方法，包括：**construct()**、destruct()、**call()**、callStatic()、**get()**、set()、**isset()**、unset()、**sleep()**、wakeup()、**toString()**、invoke()、 **set_state()**、 clone() 和 __debugInfo()，关于这些方法的用法这里不作说明，不清楚的可以翻下官方文档。

魔术方法实际是PHP提供的一些特殊操作时的钩子函数，与普通成员方法无异，它们只是与一些操作的口头约定，并没有什么字段标识它们，比如我们定义了一个函数：my_function()，我们希望在这个函数处理对象时首先调用其成员方法my_magic()，那么my_magic()也可以认为是一个魔术方法。

魔术方法与普通成员方法一样保存在 `zend_class_entry.function_table` 中，另外针对一些内核常用到的成员方法在zend_class_entry中还有一些单独的指针指向具体的成员方法：

```
struct _zend_class_entry {
    ...
    union _zend_function *constructor;
    union _zend_function *destructor;
    union _zend_function *clone;
    union _zend_function *__get;
    union _zend_function *__set;
    union _zend_function *__unset;
    union _zend_function *__isset;
    union _zend_function *__call;
    union _zend_function *__callstatic;
    union _zend_function *__tostring;
    union _zend_function *__debugInfo;
    ...
}
```

在编译成员方法时如果发现与这些魔术方法名称一致，则除了插入 `zend_class_entry.function_table` 哈希表以外，还会设置zend_class_entry中对应的指针。



具体在编译成员方法时设置：zend_begin_method_decl()。

```
void zend_begin_method_decl(zend_op_array *op_array, zend_string *name, zend_bool has_body)
{
    ...
    //插入类的function_table中
    if (zend_hash_add_ptr(&ce->function_table, lcname, op_array) == NULL) {
        zend_error_noreturn(..);
    }

    if (!in_trait && zend_string_equals_ci(lcname, ce->name)) {
        if (!ce->constructor) {
            ce->constructor = (zend_function *) op_array;
        }
    } else if (zend_string_equals_literal(lcname, ZEND_CONSTRUCTOR_FUNC_NAME)) {
        ce->constructor = (zend_function *) op_array;
    } else if (zend_string_equals_literal(lcname, ZEND_DESTRUCTOR_FUNC_NAME)) {
        ce->destructor = (zend_function *) op_array;
    } else if (zend_string_equals_literal(lcname, ZEND_CLONE_FUNC_NAME)) {
        ce->clone = (zend_function *) op_array;
    } else if (zend_string_equals_literal(lcname, ZEND_CALL_FUNC_NAME)) {
        ce->__call = (zend_function *) op_array;
    } else if (zend_string_equals_literal(lcname, ZEND_CALLSTATIC_FUNC_NAME)) {
        ce->__callstatic = (zend_function *) op_array;
    } else if (...){
        ...
    }
    ...
}
```

除了这几个其它魔术方法都没有单独的指针指向，比如：**sleep()**、wakeup()，这两个主要是serialize()、unserialize()序列化、反序列化时调用的，它们是在这俩函数中写死的，我们简单看下serialize()的实现，这个函数是通过扩展提供的：

```
//file: ext/standard/var.c
PHP_FUNCTION(serialize)
{
    zval *struc;
    php_serialize_data_t var_hash;
    smart_str buf = {0};

    if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &struc) == F
AILURE) {
        return;
    }

    php_var_serialize(&buf, struc, &var_hash);
    ...
}
```

最终由 `php_var_serialize_intern()` 处理，这个函数会根据不同的类型选择不同的处理方式：

```
static void php_var_serialize_intern(smart_str *buf, zval *struc
, php_serialize_data_t var_hash)
{
    ...
    switch (Z_TYPE_P(struc)) {
        case IS_FALSE:
            ...
        case IS_TRUE:
            ...
        case IS_NULL:
            ...
        case IS_LONG:
            ...
    }
}
```

其中类型是对象时将先检查 `zend_class_function.function_table` 中是否定义了 `__sleep()` ，如果有的话则调用：

```
//case IS_OBJEST:
...
if (ce != PHP_IC_ENTRY && zend_hash_str_exists(&ce->function_tab
le, "__sleep", sizeof("__sleep")-1)) {
    ZVAL_STRINGL(&fname, "__sleep", sizeof("__sleep") - 1);
    //调用用户自定义的__sleep()方法
    res = call_user_function_ex(CG(function_table), struc, &fnam
e, &retval, 0, 0, 1, NULL);

    if (res == SUCCESS) {
        if (Z_TYPE(retval) != IS_UNDEF) {
            if (HASH_OF(&retval)) {
                php_var_serialize_class(buf, struc, &retval, var
_hash);
            } else {
                smart_str_appendl(buf,"N;", 2);
            }
            zval_ptr_dtor(&retval);
        }
        return;
    }
}
//后面会走到IS_ARRAY分支继续序列化处理
...
```

其它魔术方法与__sleep()类似，都是在一些特殊操作中固定调用的。

3.4.6 类的自动加载

在实际使用中，通常会把一个类定义在一个文件中，然后使用时include加载进来，这样就带来一个问题：在每个文件的头部都需要包含一个长长的include列表，而且当文件名称修改时也需要把每个引用的地方都改一遍，另外前面我们也介绍过，原则上父类需要在子类定义之前定义，当存在大量类时很难得到保证，因此PHP提供了一种类的自动加载机制，当使用未被定义的类时自动调用类加载器将类加载进来，方便类的同一管理。

在内核实现上类的自动加载实际就是定义了一个钩子函数，实例化类时如果在EG(class_table)中没有找到对应的类则会调用这个钩子函数，调用完以后再重新查找一次。这个钩子函数保存在EG(autoload_func)中。

PHP中提供了两种方式实现自动加载：`__autoload()`、`spl_autoload_register()`。

(1)__autoload():

这种方式比较简单，用户自定义一个`__autoload()`函数即可，参数是类名，当实例化一个类是如果没有找到这个类则会查找用户是否定义了`__autoload()`函数，如果定义了则调用此函数，比如：

```
//文件1：my_class.php
<?php
class my_class {
    public $id = 123;
}

//文件2：b.php
<?php
function __autoload($class_name){
    //do something...
    include $class_name . '.php';
}

$obj = new my_class();
var_dump($obj);
```

(2)spl_autoload_register():

相比 `__autoload()` 只能定义一个加载器， `spl_autoload_register()` 提供了更加灵活的注册方式，可以支持任意数量的加载器，比如第三方库加载规则不可能保持一致，这样就可以通过此函数注册自己的加载器了，在实现上spl创建了一个队列来保存用户注册的加载器，然后定义了一个spl_autoload函数到EG(autoload_func)，当找不到类时内核回调spl_autoload，这个函数再依次调用用户注册的加载器，没调用一个重新检查下查找的类是否在EG(class_table)中已经注册，仍找不到的话继续调用下一个加载器，直到类成功注册为止。

```
bool spl_autoload_register ([ callable $autoload_function [, bool
 $throw = true [, bool $prepend = false ]]] )
```

参数 `$autoload_function` 为加载器，可以是函数名，第2个参数 `$throw` 用于设置autoload_function 无法成功注册时， spl_autoload_register()是否抛出异常，最后一个参数如果为true时spl_autoload_register() 会添加函数到队列之首，而不是队列尾部。

```
function autoload_one($class_name){
    echo "autoload_one->", $class_name, "\n";
}

function autoload_two($class_name){
    echo "autoload_two->", $class_name, "\n";
}

spl_autoload_register("autoload_one");
spl_autoload_register("autoload_two");

$obj = new my_class();
var_dump($obj);
```

这个例子执行时就会将autoload_one()、autoload_two()都调一遍，假如第一个函数就成功注册了my_class类则不会再调后面的加载器。

内核查找类通过 `zend_lookup_class_ex()` 完成，我们简单看下其处理过程。

```
//file: zend_execute_API.c
ZEND_API zend_class_entry *zend_lookup_class_ex(zend_string *nam
e, const zval *key, int use_autoload)
{
    ...
    //从EG(class_table)符号表找类的zend_class_entry，如果找到说明类已
经编译，直接返回
    ce = zend_hash_find_ptr(EG(class_table), lc_name);
    if (ce) {
        if (!key) {
            zend_string_release(lc_name);
        }
        return ce;
    }
    ...
    //如果没有通过spl注册则看下是否定义了__autoload()
    if (!EG(autoload_func)) {
        zend_function *func = zend_hash_str_find_ptr(EG(function
_table), "__autoload", sizeof("__autoload") - 1);
        if (func) {
            EG(autoload_func) = func;
        } else {
            return NULL;
        }
    }
    ...
    fcall_cache.function_handler = EG(autoload_func);
    ...
    //调用EG(autoload_func)函数，然后再查一次EG(class_table)
    if ((zend_call_function(&fcall_info, &fcall_cache) == SUCCES
S) && !EG(exception)) {
        ce = zend_hash_find_ptr(EG(class_table), lc_name);
    }
    ...
}
```

SPL的具体实现比较简单，这里不再介绍。

3.5 运行时缓存

在本节开始之前我们先分析一个例子：

```
class my_class {
    public $id = 123;

    public function test() {
        echo $this->id;
    }
}

$obj = new my_class;
$obj->test();
$obj->test();
...
```

这个例子定义了一个类，然后多次调用同一个成员方法，这个成员方法功能很简单：输出一个成员属性，根据前面对成员属性的介绍可以知道其查找过程为："首先根据对象找到所属zend_class_entry，然后再根据属性名查找 `zend_class_entry.properties_info` 哈希表，得到 `zend_property_info` ，最后根据属性结构的offset定位到属性值的存储位置"，概括一下这个过程就是：zend_object->zend_class_entry->properties_info->属性值，那么问题来了：每次执行 `my_class::test()` 时难道上面的过程都要完整走一遍吗？

我们再仔细看下这个过程，字面量"id"在"$this->id"此条语句中就是用来索引属性的，不管执行多少次它的任务始终是这个，那么有没有一种办法将"id"与查找到的zend_class_entry、zend_property_info.offset建立一种关联关系保存下来，这样再次执行时直接根据"id"拿到前面关联的这两个数据，从而避免多次重复相同的工作呢？这就是本节将要介绍的内容：运行时缓存。

在执行期间，PHP经常需要根据名称去不同的哈希表中查找常量、函数、类、成员方法、成员属性等，因此PHP提供了一种缓存机制用于缓存根据名称查找到的结果，以便再次执行同一opcode时直接复用上次缓存的值，无需重复查找，从而提高执行效率。

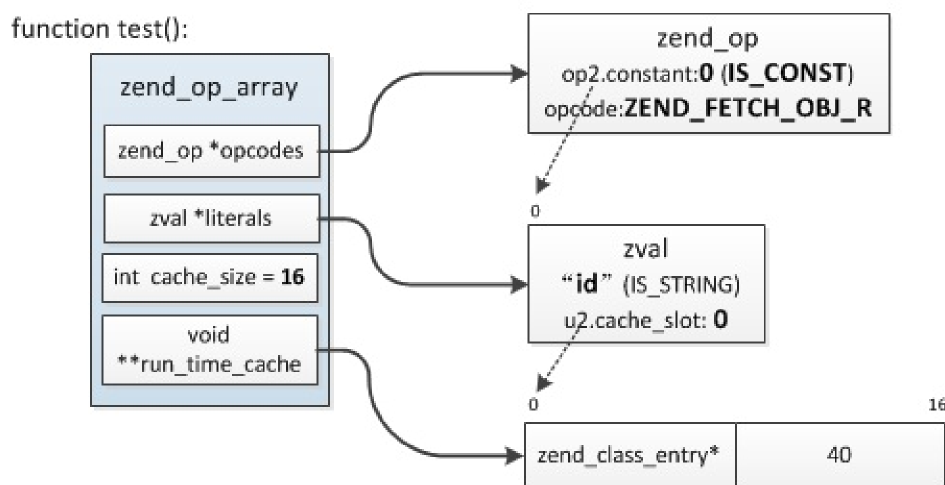
开始提到的那个例子中会缓存两个东西：zend_class_entry、zend_property_info.offset，此缓存可以认为是opcode操作的缓存，它只属于"$this->id"此语句的opcode：这样再次执行这条opcode时就直接取出上次缓存的两个值。

所以运行时缓存机制是在同一opcode执行多次的情况下才会生效，特别注意这里的同一opcode指的并不是opcode值相同，而是指内存里的同一份数据，比如： `echo $a; echo $a;` 这种就不算，因为这是两条opcode。

那么缓存是如何保存和索引的呢？执行opcode时如何知道缓存的位置？

实际上运行时缓存是基于所属opcode中CONST操作数存储的，也就是说只有包含IS_CONST类型的操作数才有可能用到此机制，其它类型都不会用到，这是因为只有CONST操作数是固定不变的，其它CV、VAR等类型值都不是固定的，既然其值是不固定的那么缓存的值也就不是固定的，所以不会针对CONST以外类型的opcode操作进行缓存，还是以开始那个例子为例，比如： `echo $this->$var;` 这种，操作数类型是CV，其正常查找时的zend_property_info是随$var值而变的，所以给他们建立一种不可变的关联关系，而： `echo $this->id;` 中"id"是固定写死的，它索引到zend_property_info始终是不变的。

缓存的存储格式是一个数组，用于保存缓存的数据指针，而指针在数组中的起始存储位置则保存在CONST操作数对应的 `zval.u2.cache_slot` 中(前面讲过，CONST操作数对应值的zval保存在zend_op_array->literals数组中)。上面那个例子对应的缓存结构：



- **(1)** 第一次执行 `echo $this->id;` 时首先根据$this取出zend_class_entry，然后根据“id”查找zend_class_entry.properties_info找到属性zend_property_info，取出此结构的offset，第一次执行后将zend_class_entry及offset保存到了test()函数的zend_op_array->run_time_cache中，占用16字

节，起始位置为0，这个值记录在“id”的zval.u2.cache_slot中；
- (2) 之后再次执行 `echo $this->id;` 时直接根据opline从zend_op_literals中取出“id”的zval，得到缓存数据保存位置：0，然后去zend_op_array->run_time_cache取出缓存的zend_class_entry、offset。

这个例子缓存数据占用了16字节(2个sizeof(void*))大小的空间，而有的只需要8字节，取决于操作类型：

- 8字节：常量、函数、类
- 16字节：成员属性、成员方法、类常量

另外一个问题是这些操作数的缓存位置(zval.u2.cache_slot)是在什么阶段确定的呢？实际上这个值是在编译阶段确定的，通过zend_op_array.cache_size记录缓存可用起始位置，编译过程中如果发现当前操作适用缓存机制，则根据缓存数据的大小从cache_size开始分配8或16字节给那个操作数，cache_size向后移动对应大小，然后将起始位置保存于CONST操作数的zval.u2.cache_slot中，执行时直接根据这个值确定缓存位置。

具体缓存的读写通过以下几个宏完成：

```
//设置缓存
CACHE_PTR(Z_CACHE_SLOT_P(EX_CONSTANT(opline->op1/2)), ptr); //ptr: 缓存的数据指针

//读取缓存
CACHED_PTR(Z_CACHE_SLOT_P(EX_CONSTANT(opline->op1/2)));

//EX_CONSTANT(opline->op1/2)是取当前IS_CONST操作数对应数据的zval
```

展开后：

```
((void**)((char*)execute_data->run_time_cache + (num)))[0]
```

`execute_data->run_time_cache` 缓存的 `zend_op_array->run_time_cache` 。

4.1 类型转换

PHP是弱类型语言，不需要明确的定义变量的类型，变量的类型根据使用时的上下文所决定，也就是变量会根据不同表达式所需要的类型自动转换，比如求和，PHP会将两个相加的值转为long、double再进行加和。每种类型转为另外一种类型都有固定的规则，当某个操作发现类型不符时就会按照这个规则进行转换，这个规则正是弱类型实现的基础。

除了自动类型转换，PHP还提供了一种强制的转换方式:

- (int)/(integer)：转换为整形 integer
- (bool)/(boolean)：转换为布尔类型 boolean
- (float)/(double)/(real)：转换为浮点型 float
- (string)：转换为字符串 string
- (array)：转换为数组 array
- (object)：转换为对象 object
- (unset)：转换为 NULL

无论是自动类型转换还是强制类型转换，不是每种类型都可以转为任意其他类型。

4.1.1 转换为NULL

这种转换比较简单，任意类型都可以转为NULL，转换时直接将新的zval类型设置为 `IS_NULL` 即可。

4.1.2 转换为布尔型

当转换为 boolean 时，根据原值的TRUE、FALSE决定转换后的结果，以下值被认为是 FALSE：

- 布尔值 FALSE 本身
- 整型值 0
- 浮点型值 0.0
- 空字符串，以及字符串 "0"
- 空数组
- NULL

所有其它值都被认为是 TRUE，比如资源、对象(这里指默认情况下，因为可以通过扩展改变这个规则)。

判断一个值是否为true的操作：

```
static zend_always_inline int i_zend_is_true(zval *op)
{
    int result = 0;

again:
    switch (Z_TYPE_P(op)) {
        case IS_TRUE:
            result = 1;
            break;
        case IS_LONG:
            //非0即真
            if (Z_LVAL_P(op)) {
                result = 1;
            }
            break;
        case IS_DOUBLE:
            if (Z_DVAL_P(op)) {
                result = 1;
            }
            break;
        case IS_STRING:
            //非空字符串及"0"外都为true
            if (Z_STRLEN_P(op) > 1 || (Z_STRLEN_P(op) && Z_STRVA
L_P(op)[0] != '0')) {
                result = 1;
            }
            break;
        case IS_ARRAY:
            //非空数组为true
            if (zend_hash_num_elements(Z_ARRVAL_P(op))) {
                result = 1;
            }
            break;
        case IS_OBJECT:
            //默认情况下始终返回true
```

```
                result = zend_object_is_true(op);
                break;
            case IS_RESOURCE:
                //合法资源就是true
                if (EXPECTED(Z_RES_HANDLE_P(op))) {
                    result = 1;
                }
            case IS_REFERENCE:
                op = Z_REFVAL_P(op);
                goto again;
                break;
            default:
                break;
        }
        return result;
}
```

在扩展中可以通过 `convert_to_boolean()` 这个函数直接将原zval转为bool型，转换时的判断逻辑与 `i_zend_is_true()` 一致。

4.1.3 转换为整型

其它类型转为整形的转换规则：

- NULL：转为0
- 布尔型：false转为0，true转为1
- 浮点型：向下取整，比如： `(int)2.8 => 2`
- 字符串：就是C语言strtoll()的规则，如果字符串以合法的数值开始，则使用该数值，否则其值为 0（零），合法数值由可选的正负号，后面跟着一个或多个数字（可能有小数点），再跟着可选的指数部分
- 数组：很多操作不支持将一个数组自动整形处理，比如： `array() + 2` ，将报error错误，但可以强制把数组转为整形，非空数组转为1，空数组转为0，没有其他值
- 对象：与数组类似，很多操作也不支持将对象自动转为整形，但有些操作只会抛一个warning警告，还是会把对象转为1操作的，这个需要看不同操作的处理情况
- 资源：转为分配给这个资源的唯一编号

具体处理：

```
ZEND_API zend_long ZEND_FASTCALL _zval_get_long_func(zval *op)
{
try_again:
    switch (Z_TYPE_P(op)) {
        case IS_NULL:
        case IS_FALSE:
            return 0;
        case IS_TRUE:
            return 1;
        case IS_RESOURCE:
            //资源将转为zend_resource->handler
            return Z_RES_HANDLE_P(op);
        case IS_LONG:
            return Z_LVAL_P(op);
        case IS_DOUBLE:
            return zend_dval_to_lval(Z_DVAL_P(op));
        case IS_STRING:
            //字符串的转换调用C语言的strtoll()处理
            return ZEND_STRTOL(Z_STRVAL_P(op), NULL, 10);
        case IS_ARRAY:
            //根据数组是否为空转为0,1
            return zend_hash_num_elements(Z_ARRVAL_P(op)) ? 1 : 0
;
        case IS_OBJECT:
            {
                zval dst;
                convert_object_to_type(op, &dst, IS_LONG, conver
t_to_long);
                if (Z_TYPE(dst) == IS_LONG) {
                    return Z_LVAL(dst);
                } else {
                    //默认情况就是1
                    return 1;
                }
            }
        case IS_REFERENCE:
            op = Z_REFVAL_P(op);
            goto try_again;
```

```
            EMPTY_SWITCH_DEFAULT_CASE()
    }
    return 0;
}
```

4.1.4 转换为浮点型

除字符串类型外，其它类型转换规则与整形基本一致，就是整形转换结果加了一位小数，字符串转为浮点数由 `zend_strtod()` 完成，这个函数非常长，定义在 `zend_strtod.c` 中，这里不作说明。

4.1.5 转换为字符串

一个值可以通过在其前面加上 (string) 或用 strval() 函数来转变成字符串。在一个需要字符串的表达式中，会自动转换为 string，比如在使用函数 echo 或 print 时，或在一个变量和一个 string 进行比较时，就会发生这种转换。

```
ZEND_API zend_string* ZEND_FASTCALL _zval_get_string_func(zval *
op)
{
try_again:
    switch (Z_TYPE_P(op)) {
        case IS_UNDEF:
        case IS_NULL:
        case IS_FALSE:
            //转为空字符串""
            return ZSTR_EMPTY_ALLOC();
        case IS_TRUE:
            //转为"1"
            ...
            return zend_string_init("1", 1, 0);
        case IS_RESOURCE: {
            //转为"Resource id #xxx"
            ...
            len = snprintf(buf, sizeof(buf), "Resource id #" ZEN
D_LONG_FMT, (zend_long)Z_RES_HANDLE_P(op));
            return zend_string_init(buf, len, 0);
```

```
        }
        case IS_LONG: {
            return zend_long_to_str(Z_LVAL_P(op));
        }
        case IS_DOUBLE: {
            return zend_strpprintf(0, "%.*G", (int) EG(precision
), Z_DVAL_P(op));
        }
        case IS_ARRAY:
            //转为"Array"，但是报Notice
            zend_error(E_NOTICE, "Array to string conversion");
            return zend_string_init("Array", sizeof("Array")-1, 0
);
        case IS_OBJECT: {
            //报Error错误
            zval tmp;
            ...
            zend_error(EG(exception) ? E_ERROR : E_RECOVERABLE_E
RROR, "Object of class %s could not be converted to string", ZST
R_VAL(Z_OBJCE_P(op)->name));
            return ZSTR_EMPTY_ALLOC();
        }
        case IS_REFERENCE:
            op = Z_REFVAL_P(op);
            goto try_again;
        case IS_STRING:
            return zend_string_copy(Z_STR_P(op));
        EMPTY_SWITCH_DEFAULT_CASE()
    }
    return NULL;
}
```

4.1.6 转换为数组

如果将一个null、integer、float、string、boolean 和 resource 类型的值转换为数组，将得到一个仅有一个元素的数组，其下标为 0，该元素即为此标量的值。换句话说，(array)$scalarValue 与 array($scalarValue) 完全一样。

如果一个 object 类型转换为 array，则结果为一个数组，数组元素为该对象的全部属性，包括public、private、protected，其中private的属性转换后的key加上了类名作为前缀，protected属性的key加上了"*"作为前缀，但是这个前缀并不是转为数组时单独加上的，而是类编译生成属性zend_property_info时就已经加上了，也就是说这其实是成员属性本身的一个特点，举例来看：

```
class test {
    private $a = 123;
    public $b = "bbb";
    protected $c = "ccc";
}
$obj = new test;
print_r((array)$obj);
======================
Array
(
    [testa] => 123
    [b] => bbb
    [*c] => ccc
)
```

转换时的处理：

```
ZEND_API void ZEND_FASTCALL convert_to_array(zval *op)
{
    try_again:
    switch (Z_TYPE_P(op)) {
        case IS_ARRAY:
            break;
        case IS_OBJECT:
            ...
            if (Z_OBJ_HT_P(op)->get_properties) {
                //获取所有属性数组
                HashTable *obj_ht = Z_OBJ_HT_P(op)->get_properti
es(op);
                //将数组内容拷贝到新数组
                ...
            }
        case IS_NULL:
```

```
            ZVAL_NEW_ARR(op);
            //转为空数组
            zend_hash_init(Z_ARRVAL_P(op), 8, NULL, ZVAL_PTR_DTO
R, 0);
            break;
        case IS_REFERENCE:
            zend_unwrap_reference(op);
            goto try_again;
        default:
            convert_scalar_to_array(op);
            break;
    }
}

//其他标量类型转array
static void convert_scalar_to_array(zval *op)
{
    zval entry;

    ZVAL_COPY_VALUE(&entry, op);
    //新分配一个数组，将原值插入数组
    ZVAL_NEW_ARR(op);
    zend_hash_init(Z_ARRVAL_P(op), 8, NULL, ZVAL_PTR_DTOR, 0);
    zend_hash_index_add_new(Z_ARRVAL_P(op), 0, &entry);
}
```

4.1.7 转换为对象

如果其它任何类型的值被转换成对象，将会创建一个内置类 stdClass 的实例：如果该值为 NULL，则新的实例为空；array转换成object将以键名成为属性名并具有相对应的值，数值索引的元素也将转为属性，但是无法通过"->"访问，只能遍历获取；对于其他值，会以scalar作为属性名。

```
ZEND_API void ZEND_FASTCALL convert_to_object(zval *op)
{
try_again:
    switch (Z_TYPE_P(op)) {
        case IS_ARRAY:
            {
                HashTable *ht = Z_ARR_P(op);
                ...
                //以key为属性名，将数组元素拷贝到对象属性
                object_and_properties_init(op, zend_standard_cla
ss_def, ht);
                break;
            }
        case IS_OBJECT:
            break;
        case IS_NULL:
            object_init(op);
            break;
        case IS_REFERENCE:
            zend_unwrap_reference(op);
            goto try_again;
        default: {
            zval tmp;
            ZVAL_COPY_VALUE(&tmp, op);
            object_init(op);
            //以scalar作为属性名
            zend_hash_str_add_new(Z_OBJPROP_P(op), "scalar", siz
eof("scalar")-1, &tmp);
            break;
        }
    }
}
```

4.1.8 转换为资源

无法将其他类型转为资源。

4.2 选择结构

程序并不都是顺序执行的，选择结构用于判断给定的条件，根据判断的结果来控制程序的流程。PHP中通过if、elseif、else和switch语句实现条件控制。这一节我们就分析下PHP中两种条件语句的具体实现。

4.2.1 if语句

If语句用法：

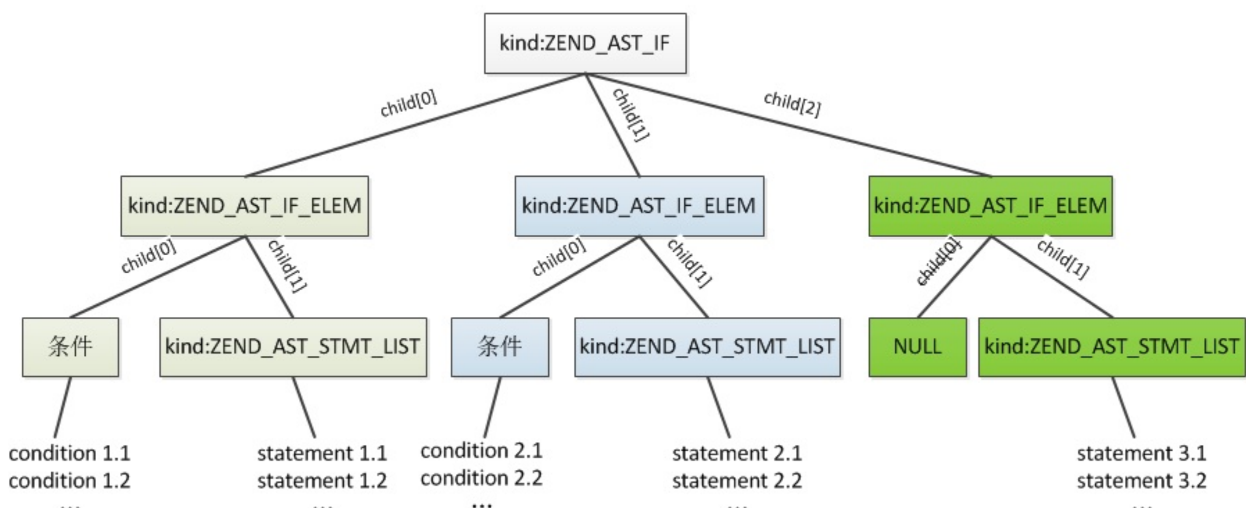
```
if(Condition1){
    Statement1;
}elseif(Condition2){
    Statement2;
}else{
    Statement3;
}
```

IF语句有两部分组成：condition(条件)、statement(声明)，每个条件分支对应一组这样的组合，其中最后的else比较特殊，它没有条件，编译时也是按照这个逻辑编译为一组组的condition和statement，其具体的语法规则如下：

```
if_stmt:
        if_stmt_without_else %prec T_NOELSE { $$ = $1; }
    |   if_stmt_without_else T_ELSE statement
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AS
T_IF_ELEM, NULL, $3)); }
;

if_stmt_without_else:
        T_IF '(' expr ')' statement { $$ = zend_ast_create_list(1
, ZEND_AST_IF,
                                        zend_ast_create(ZEND_AST
_IF_ELEM, $3, $5)); }
    |   if_stmt_without_else T_ELSEIF '(' expr ')' statement
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AS
T_IF_ELEM, $4, $6)); }
;
```

从上面的语法规则可以看出，编译if语句时首先会创建一个 `ZEND_AST_IF` 的节点，这个节点是一个list，用于保存各个分支的condition、statement，编译每个分支时将创建一个 `ZEND_AST_IF_ELEM` 的节点，它有两个子节点，分别用来记录：condition、statement，然后把这个节点插入到 `ZEND_AST_IF` 下，最终生成的AST：



编译opcode时顺序编译每个分支的condition、statement即可，编译过程大致如下：

- **(1)** 编译当前分支的condition语句，这里可能会有多个条件，但最终会归并为

一个true/false的结果；

- **(2)** 编译完condition后编译一条ZEND_JMPZ的opcode，这条opcode用来判断当前condition最终为true还是false，如果当前condition成立直接继续执行本组statement即可，无需进行跳转，但是如果不成立就需要跳过本组的statement，所以这条opcode还需要知道该往下跳过多少条opcode，而跳过的这些opcode就是本组的statement，因此这个值需要在编译完本组statement后才能确定，现在还无法确定；
- **(3)** 编译当前分支的statement列表，其节点类型ZEND_AST_STMT_LIST，就是普通语句的编译；
- **(4)** 编译完statement后编译一条ZEND_JMP的opcode，这条opcode是当condition成立执行完本组statement时跳出if的，因为当前分支既然条件成立就不需要再跳到其他分支，执行完当前分支的statement后将直接跳出if，所以ZEND_JMP需要知道该往下跳过多少opcode，而跳过的这些opcode是后面所有分支的opcode数，只有编译完全部分支后才能确定；
- **(5)** 编译完statement后再设置步骤(2)中条件不成立时ZEND_JMPZ应该跳过的opcode数；
- **(6)** 重复上面的过程依次编译后面的condition、statement，编译完全部分支后再设置各分支在步骤(4)中ZEND_JMP跳出if的opcode位置。

具体的编译过程在 `zend_compile_if()` 中，过程比较清晰:

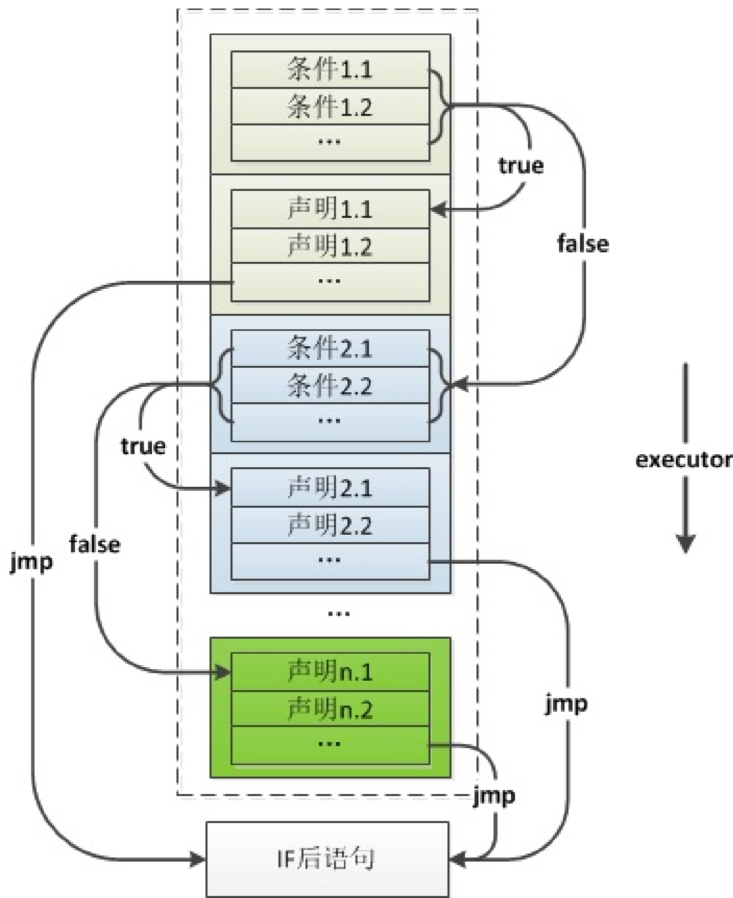
```
void zend_compile_if(zend_ast *ast)
{
    zend_ast_list *list = zend_ast_get_list(ast);
    uint32_t i;
    uint32_t *jmp_opnums = NULL;

    //用来保存每个分支在步骤(4)中的ZEND_JMP opcode
    if (list->children > 1) {
        jmp_opnums = safe_emalloc(sizeof(uint32_t), list->childr
en - 1, 0);
    }
    //依次编译各个分支
    for (i = 0; i < list->children; ++i) {
        zend_ast *elem_ast = list->child[i];
        zend_ast *cond_ast = elem_ast->child[0]; //条件
        zend_ast *stmt_ast = elem_ast->child[1]; //声明
```

```
        znode cond_node;
        uint32_t opnum_jmpz;
        if (cond_ast) {
            //编译condition
            zend_compile_expr(&cond_node, cond_ast);
            //编译condition跳转opcode：ZEND_JMPZ
            opnum_jmpz = zend_emit_cond_jump(ZEND_JMPZ, &cond_no
de, 0);
        }
        //编译statement
        zend_compile_stmt(stmt_ast);
        //编译statement执行完后跳出if的opcode：ZEND_JMP(最后一个分支无
需这条opcode)
        if (i != list->children - 1) {
            jmp_opnums[i] = zend_emit_jump(0);
        }
        if (cond_ast) {
            //设置ZEND_JMPZ跳过opcode数
            zend_update_jump_target_to_next(opnum_jmpz);
        }
    }

    if (list->children > 1) {
        //设置前面各分支statement执行完后应跳转的位置
        for (i = 0; i < list->children - 1; ++i) {
            zend_update_jump_target_to_next(jmp_opnums[i]); //设
置每组stmt最后一条jmp跳转为if外
        }
        efree(jmp_opnums);
    }
}
```

最终if语句编译后基本是这样的结构：



执行时依次判断各分支条件是否成立，成立则执行当前分支statement，执行完后跳到if外语句；不成立则调到下一分支继续判断是否成立，以此类推。不管各分支条件有几个，其最终都会归并为一个结果，也就是每个分支只需要判断最终的条件值是否为true即可，而多个条件计算得到最终值的过程就是普通的逻辑运算。

> **Note:** 注意elseif与else if，上面介绍的是elseif的编译，而else if则实际相当于嵌套了一个if，也就是说一个if的分支中包含了另外一个if，在编译、执行的过程中这两个是有差别的。

4.2.2 switch语句

switch语句与if类似，都是条件语句，很多时候需要将一个变量或者表达式与不同的值进行比较，根据不同的值执行不同的代码，这种场景下用if、switch都可以实现，但switch相对更加直观。

switch语法：

```
switch(expression){
    case value1:
        statement1;
    case value2:
        statement2;
    ...
    default:
        statementn;
}
```

这里并没有将break加入到switch的语法中，因为严格意义上break并不是switch的一部分，break属于另外一类单独的语法：中断语法，PHP中如果没有在switch中加break则执行时会从命中的那个case开始一直执行到结束，这与很多其它的语言不同(比如：golang)。

从switch的语法可以看出，switch主要包含两部分：expression、case list，case list包含多个case，每个case包含value、statement两部分。expression是一个表达式，但它将在case对比前执行，所以switch最终执行时就是拿expression的值逐个与case的value比较，如果相等则从命中case的statement开始向下执行。

下面看下switch的语法规则：

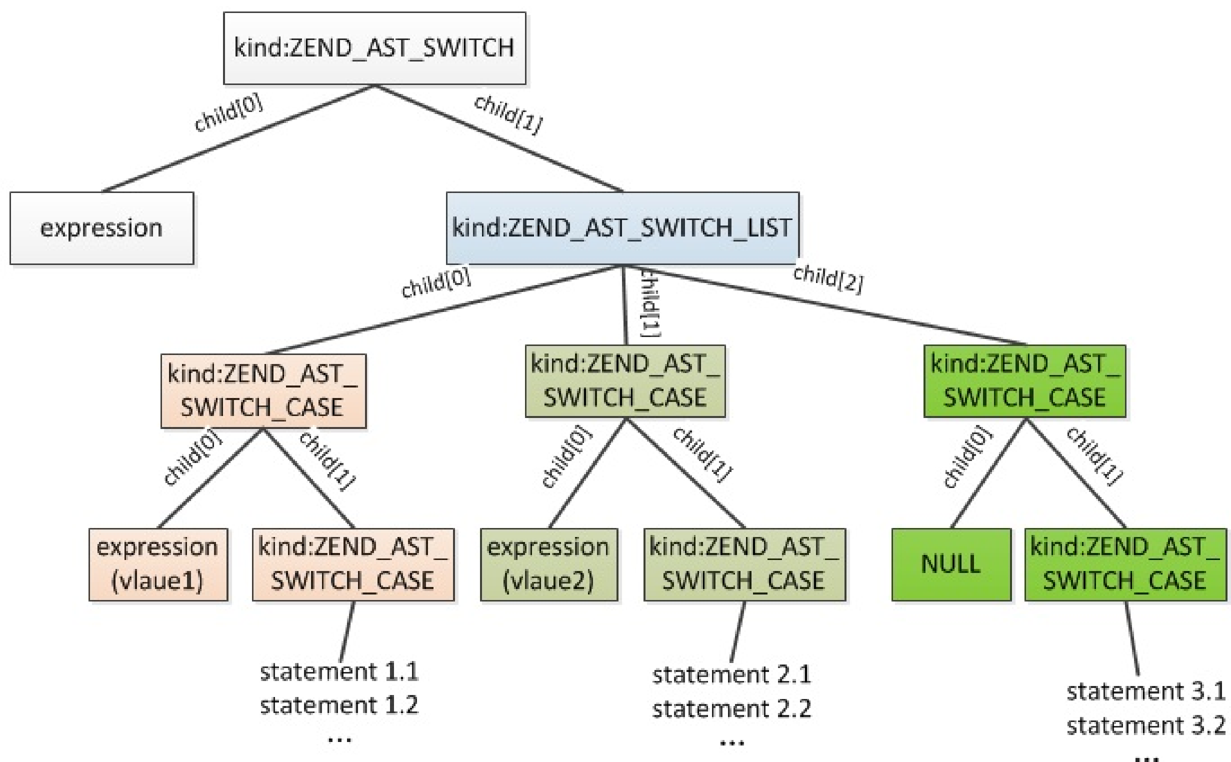
```
statement:
    ...
    |   T_SWITCH '(' expr ')' switch_case_list { $$ = zend_ast_c
reate(ZEND_AST_SWITCH, $3, $5); }
    ...
;

switch_case_list:
        '{' case_list '}'                   { $$ = $2; }
    |   '{' ';' case_list '}'               { $$ = $3; }
    |   ':' case_list T_ENDSWITCH ';'       { $$ = $2; }
    |   ':' ';' case_list T_ENDSWITCH ';'   { $$ = $3; }
;

case_list:
        /* empty */ { $$ = zend_ast_create_list(0, ZEND_AST_SWIT
CH_LIST); }
    |   case_list T_CASE expr case_separator inner_statement_lis
t
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AS
T_SWITCH_CASE, $3, $5)); }
    |   case_list T_DEFAULT case_separator inner_statement_list
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AS
T_SWITCH_CASE, NULL, $4)); }
;

case_separator:
        ':'
    |   ';'
;
```

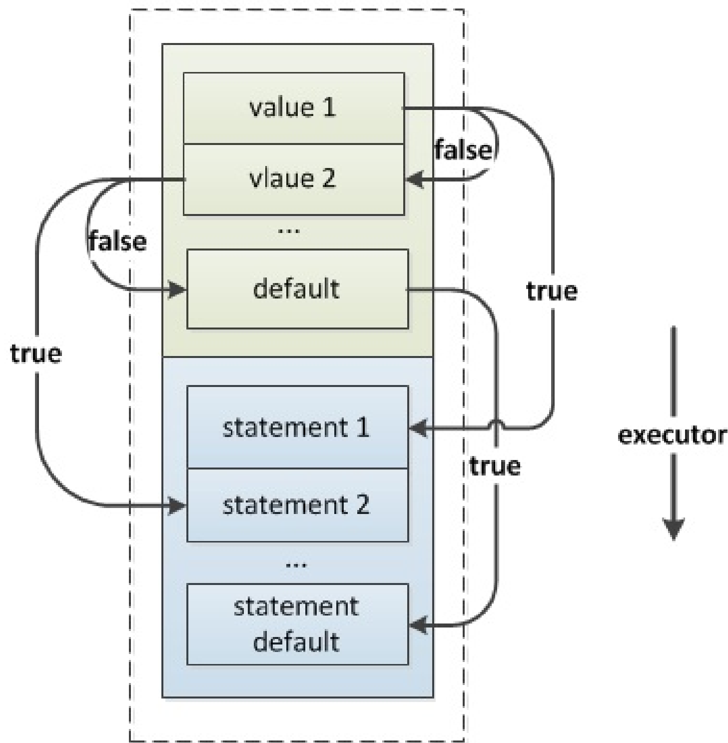
从语法解析规则可以看出，switch最终被解析为一个 `ZEND_AST_SWITCH` 节点，这个节点主要包含两个子节点：expression、case list，其中expression节点比较简单，case list节点对应一个 `ZEND_AST_SWITCH_LIST` 节点，这个节点是一个list，有多个case子节点，每个case节点对应一个 `ZEND_AST_SWITCH_CASE` 节点，包括value（或expr）、statement两个子节点，生成的AST如下：



与if不同，switch不会像if那样依次把每个分支编译为一组组的condition、statement，而是会先编译全部case的value表达式，再编译全部case的statement，编译过程大致如下：

- (1)首先编译expression，其最终将得到一个固定的value；
- (2)依次编译每个case的value，如果value是一个表达式则编译expression，与(1)相同，执行时其最终也是一个固定的value，每个case编译一条ZEND_CASE的opcode，除了这条opcode还会编译出一条ZEND_JMPNZ的opcode，这条opcode用来跳到当前case的statement的开始位置，但是statement在这时还未编译，所以ZEND_JMPNZ的跳转值暂不确定；
- (3)编译完全部case的value后接着从头开始编译每个case的statement，编译前首先设置步骤(2)中ZEND_JMPNZ的跳转值为当前statement起始位置。

具体编译过程在 `zend_compile_switch()` 中，这里不再展开，编译后的基本结构如下：



执行时首先如果switch的是一个表达式则会首先执行表达式的语句，然后再拿最终的结果逐个与case的值比较，如果case也是一个表达式则也先执行表达式，执行完再与switch的值比较，比较结果如果为true则跳到当前case的statement位置开始顺序执行，如果结果为false则继续向下执行，与下一个case比较，以此类推。

> **Note:**
>
> **(1)** case不管是表达式还是固定的值其最终比较时是一样的，如果是表达式则将其执行完以后再作比较，也就是说switch并不支持case多个值的用法，比如：case value1 || value2 : statement，这么写首先是会执行(value1 || value2)，然后把结果与switch的值比较，并不是指switch的值等于value1或value2，这个地方一定要注意，如果想命中多个value只能写到不同case下
>
> **(2)** switch的value与case的value比较用的是"=="，而不是"==="

4.3 循环结构

实际应用中有许多具有规律性的重复操作，因此在程序中就需要重复执行某些语句。循环结构是在一定条件下反复执行某段程序的流程结构，被反复执行的程序称为循环体。循环语句是由循环体及循环的终止条件两部分组成的。

PHP中的循环结构有4种：while、for、foreach、do while，接下来我们分析下这几个结构的具体的实现。

4.3.1 while循环

while循环的语法：

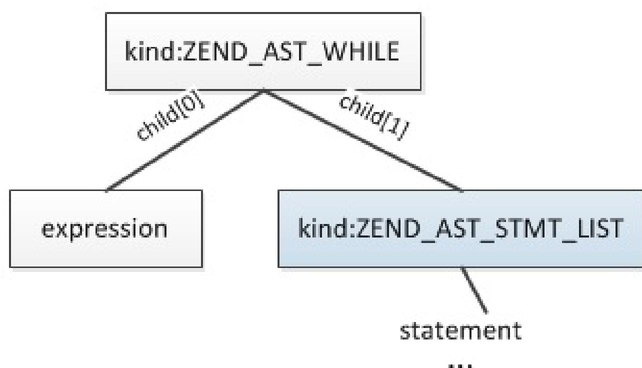
```
while(expression)
{
    statement;//循环体
}
```

while的结构比较简单，由两部分组成：expression、statement，其中expression为循环判断条件，当expression为true时重复执行statement，具体的语法规则：

```
statement:
    ...
    |   T_WHILE '(' expr ')' while_statement { $$ = zend_ast_create(ZEND_AST_WHILE, $3, $5); }
    ...
;

while_statement:
        statement { $$ = $1; }
    |   ':' inner_statement_list T_ENDWHILE ';' { $$ = $2; }
;
```

从while语法规则可以看出，在解析时会创建一个 `ZEND_AST_WHILE` 节点，expression、statement分别保存在两个子节点中，其AST如下：



while编译的过程也比较简单，比较特别的是while首先编译的是循环体，然后才是循环判断条件，更像是do while，编译过程大致如下：

- **(1)** 首先编译一条ZEND_JMP的opcode，这条opcode用来跳到循环判断条件expression的位置，由于while是先编译循环体再编译循环条件，所以此时还无法确定具体的跳转值；
- **(2)** 编译循环体statement；编译完成后更新步骤(1)中ZEND_JMP的跳转值；
- **(3)** 编译循环判断条件expression；
- **(4)** 编译一条ZEND_JMPNZ的opcode，这条opcode用于循环判断条件执行完以后跳到循环体的，如果循环条件成立则通过此opcode跳到循环体开始的位置，否则继续往下执行(即：跳出循环)。

具体的编译过程：

```
void zend_compile_while(zend_ast *ast)
{
    zend_ast *cond_ast = ast->child[0];
    zend_ast *stmt_ast = ast->child[1];
    znode cond_node;
    uint32_t opnum_start, opnum_jmp, opnum_cond;

    //(1)编译ZEND_JMP
    opnum_jmp = zend_emit_jump(0);

    zend_begin_loop(ZEND_NOP, NULL);

    //(2)编译循环体statement，opnum_start为循环体起始位置
    opnum_start = get_next_op_number(CG(active_op_array));
    zend_compile_stmt(stmt_ast);

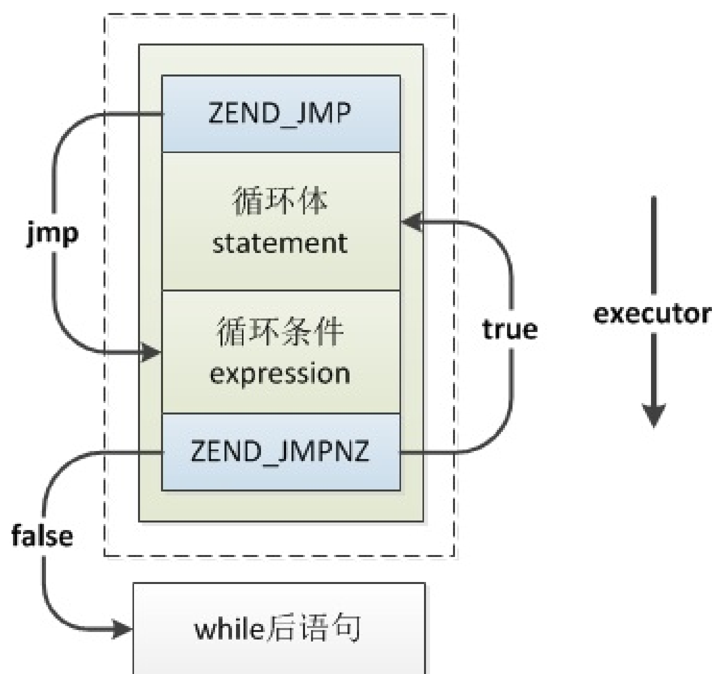
    //设置ZEND_JMP opcode的跳转值
    opnum_cond = get_next_op_number(CG(active_op_array));
    zend_update_jump_target(opnum_jmp, opnum_cond);

    //(3)编译循环条件expression
    zend_compile_expr(&cond_node, cond_ast);

    //(4)编译ZEND_JMPNZ，用于循环条件成立时跳回循环体开始位置：opnum_st
art
    zend_emit_cond_jump(ZEND_JMPNZ, &cond_node, opnum_start);

    zend_end_loop(opnum_cond);
}
```

编译后opcode整体如下：



运行时首先执行 `ZEND_JMP` ，跳到while条件expression处开始执行，然后由 `ZEND_JMPNZ` 对条件的执行结果进行判断，如果条件成立则跳到循环体statement起始位置开始执行，如果条件不成立则继续向下执行，跳出while，第一次循环执行以后将不再执行 `ZEND_JMP` ，后续循环只有靠 `ZEND_JMPNZ` 控制跳转，循环体执行完成后接着执行循环判断条件，进行下一轮循环的判断。

> **Note:** 实际执行时可能会省略 `ZEND_JMPNZ` 这一步，这是因为很多while条件expression执行完以后会对下一条opcode进行判断，如果是 `ZEND_JMPNZ` 则直接根据条件成立与否进行快速跳转，不需要再由 `ZEND_JMPNZ` 判断，比如：
>
> $a = 123; while($a > 100){ echo "yes"; } `$a > 100` 对应的opcode：ZEND_IS_SMALLER，执行时发现$a与100类型可以直接比较(都是long)，则直接就能知道循环条件的判断结果，这种情况下将会判断下一条opcode是否为ZEND_JMPNZ，是的话直接设置下一条要执行的opcode，这样就不需要再单独执行依次ZEND_JMPNZ了。
>
> 上面的例子如果 `$a = '123';` 就不会快速进行处理了，而是按照正常的逻辑调用ZEND_JMPNZ。

4.3.2 do while循环

do while与while非常相似，唯一的区别在于do while第一次执行时不需要判断循环条件。

do while循环的语法：

```
do{
    statement;//循环体
}while(expression)
```

do while编译过程与while的基本一致，不同的地方在于do while没有 `ZEND_JMP` 这条opcode：

```
void zend_compile_do_while(zend_ast *ast)
{
    zend_ast *stmt_ast = ast->child[0];
    zend_ast *cond_ast = ast->child[1];

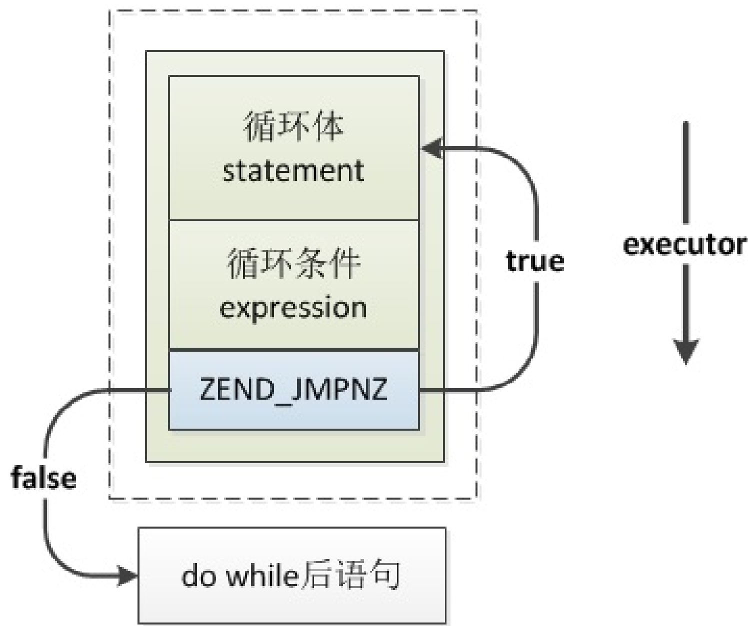
    znode cond_node;
    uint32_t opnum_start, opnum_cond;

    //(1)编译循环体statement，opnum_start为循环体起始位置
    opnum_start = get_next_op_number(CG(active_op_array));
    zend_compile_stmt(stmt_ast);

    //(2)编译循环判断条件expression
    opnum_cond = get_next_op_number(CG(active_op_array));
    zend_compile_expr(&cond_node, cond_ast);

    //(3)编译ZEND_JMPNZ
    zend_emit_cond_jump(ZEND_JMPNZ, &cond_node, opnum_start);
}
```

编译后的结果：



运行时首先执行循环体statement，然后执行循环判断条件，如果条件成立跳到循环体起始位置，否则结束循环。

4.3.3 for循环

for循环语法：

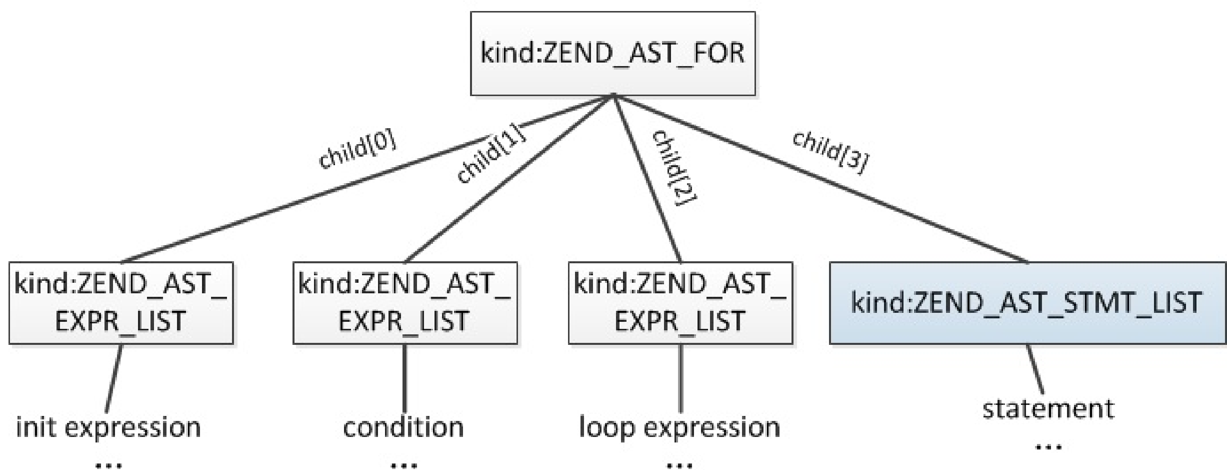
```
for (init expr; condition expr; loop expr){
    statement
}
```

init expr在循环开始前无条件执行一次，后面循环不再执行；condition expr在每次循环开始前运算，是循环的判断条件，如果值为true，则继续循环，执行循环体，如果值为false，则终止循环；loop expr在每次循环体执行完以后被执行。

for的语法规则：

```
statement:
    ...
    |   T_FOR '(' for_exprs ';' for_exprs ';' for_exprs ')' for_
statement
            { $$ = zend_ast_create(ZEND_AST_FOR, $3, $5, $7, $9)
; }
    ...
;
```

从语法规则可以看出，for被编译为 `ZEND_AST_FOR` 节点，包含4个子节点，分别为：expr1、expr2、expr3、statement。



for的编译与while类似，只是多了init expr、loop expr两部分，编译过程大致如下：

- **(1)** 首先编译初始化表达式：init expr;
- **(2)** 编译一条 `ZEND_JMP` 的opcode，此opcode用于跳到条件expression位置，具体跳转值需要后面才能确定；
- **(3)** 编译循环体statement；
- **(4)** 编译loop expr；然后设置步骤(2)中 `ZEND_JMP` 的跳转值；
- **(5)** 编译循环条件：condition expr；
- **(6)** 编译一条 `ZEND_JMPNZ` ，此opcode用于循环条件成立时跳到循环体起始位置。

具体编译过程：

```
void zend_compile_for(zend_ast *ast)
{
    zend_ast *init_ast = ast->child[0];
    zend_ast *cond_ast = ast->child[1];
    zend_ast *loop_ast = ast->child[2];
    zend_ast *stmt_ast = ast->child[3];

    znode result;
    uint32_t opnum_start, opnum_jmp, opnum_loop;

    //(1)编译init expression
    zend_compile_expr_list(&result, init_ast);
    zend_do_free(&result);

    //(2)编译ZEND_JMP
    opnum_jmp = zend_emit_jump(0);

    //opnum_start是循环体起始位置
    opnum_start = get_next_op_number(CG(active_op_array));

    //(3)编译循环体
    zend_compile_stmt(stmt_ast);

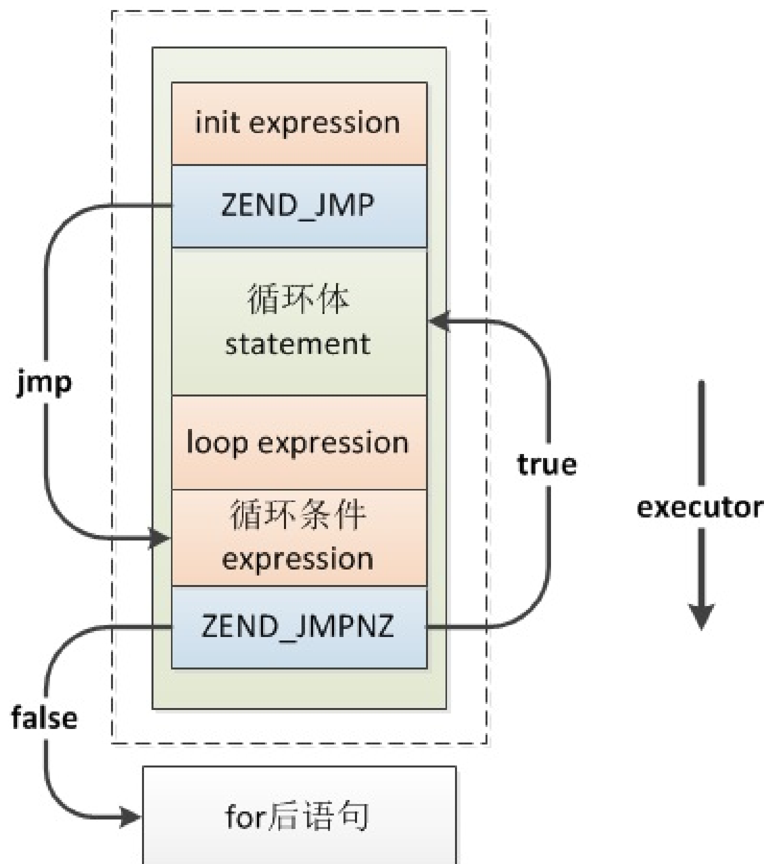
    //(4)编译loop expression
    opnum_loop = get_next_op_number(CG(active_op_array));
    zend_compile_expr_list(&result, loop_ast);
    zend_do_free(&result);

    //设置ZEND_JMP跳转值
    zend_update_jump_target_to_next(opnum_jmp);

    //(5)编译循环条件expression
    zend_compile_expr_list(&result, cond_ast);
    zend_do_extended_info();

    //(6)编译ZEND_JMPNZ
    zend_emit_cond_jump(ZEND_JMPNZ, &result, opnum_start);
}
```

最终编译结果：



运行时首先执行初始化表达式：init expression，然后执行 `ZEND_JMP` 跳到循环条件expression处，如果条件成立则执行 `ZEND_JMPNZ` 跳到循环体起始位置依次执行循环体、loop expression，如果条件不成立则终止循环，第一次循环之后就是： `循环条件->ZEND_JMPNZ->循环体->loop expression` 之间循环了。

4.3.4 foreach循环

foreach是PHP针对数组、对象提供的一种遍历方式，foreach语法：

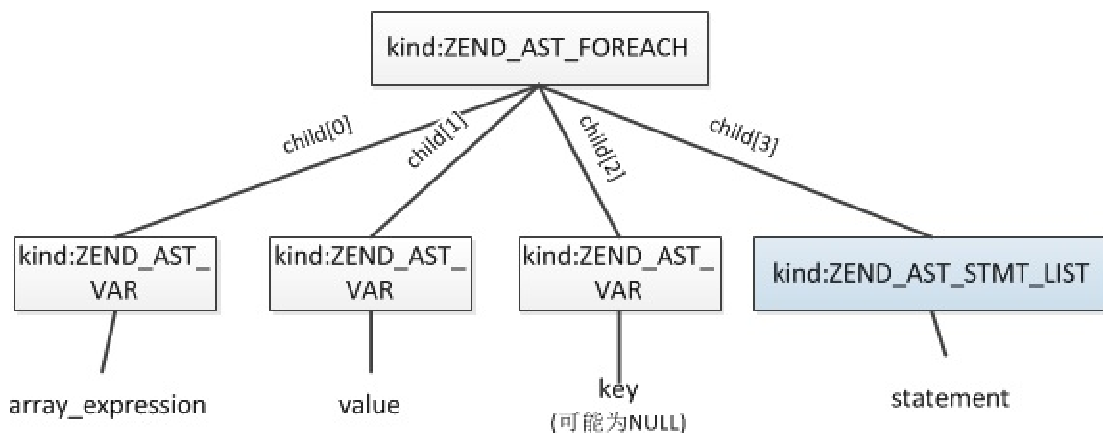
```
foreach (array_expression as $key => $value){
    statement
}
```

遍历arraiy_expression时每次循环会把当前单元的值赋给$value，当前单元的键值赋给$key，其中$key可以省略，$value前也可以加"&"表示引用单元的值。

foreach的语法规则：

```
statement:
    ...
    //省略key的规则: foreach($array as $v){ ... }
    |   T_FOREACH '(' expr T_AS foreach_variable ')' foreach_sta
tement
            { $$ = zend_ast_create(ZEND_AST_FOREACH, $3, $5, NULL
, $7); }
    //有key的规则: foreach($array as $k=>$v){ ... }
    |   T_FOREACH '(' expr T_AS foreach_variable T_DOUBLE_ARROW
foreach_variable ')' foreach_statement
            { $$ = zend_ast_create(ZEND_AST_FOREACH, $3, $7, $5,
 $9); }
    ...
;
```

foreach在编译阶段解析为 `ZEND_AST_FOREACH` 节点，包含4个子节点，分别表示：遍历的数组或对象、遍历的value、遍历的key以及循环体，生成的AST类似这样：



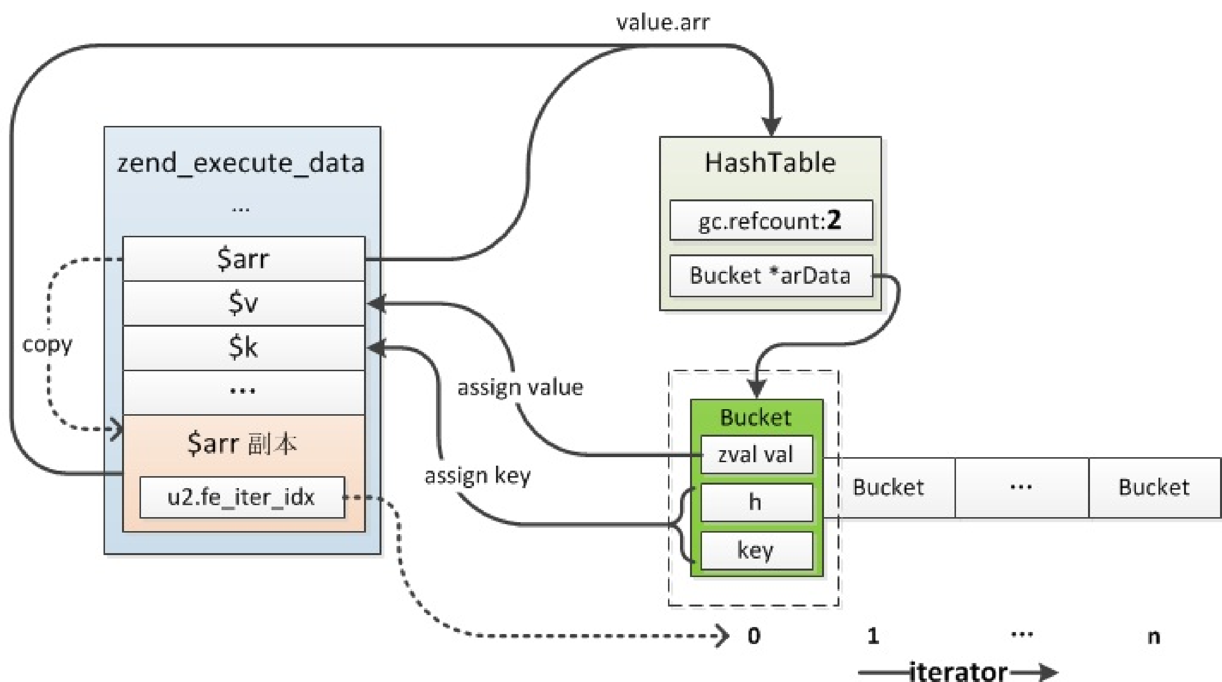
如果value是指向数组或对象成员的引用，则value对应的节点类型为 `ZEND_AST_REF` 。

相对上面几种常规的循环结构，foreach的实现略显复杂：$key、$value实际就是两个普通的局部变量，遍历的过程就是对两个局部变量不断赋值、更新的过程，以数组为例，首先将数组拷贝一份用于遍历(只拷贝zval，value还是指向同一份)，从arData第一个元素开始，把Bucket.zval.value值赋值给$value，把Bucket.key(或Bucket.h)赋值给$key，然后更新迭代位置：将下一个元素的位置记录

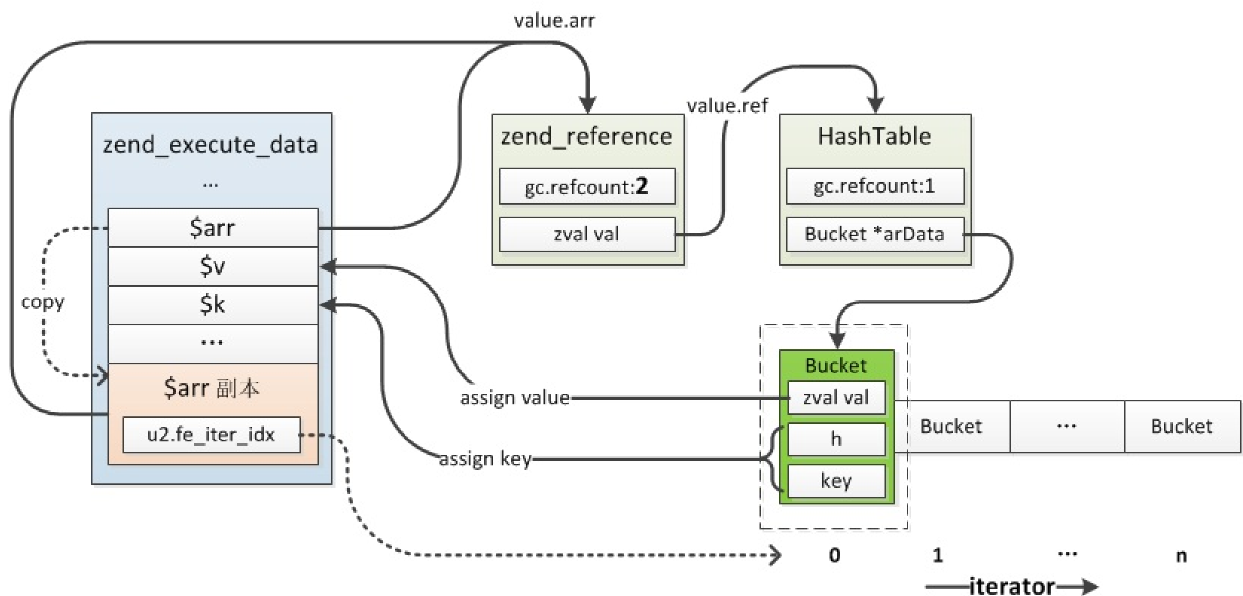
在 `zval.u2.fe_iter_idx` 中，这样下一轮遍历时直接从这个位置开始，这也是遍历前为什么要拷贝一份zval用于遍历的原因，如果发现 `zval.u2.fe_iter_idx` 已经到达arData末尾了则结束遍历，销毁一开始拷贝的zval。举个例子来看：

```
$arr = array(1,2,3);
foreach($arr as $k=>$v){
    echo $v;
}
```

局部变量对应的内存结构：



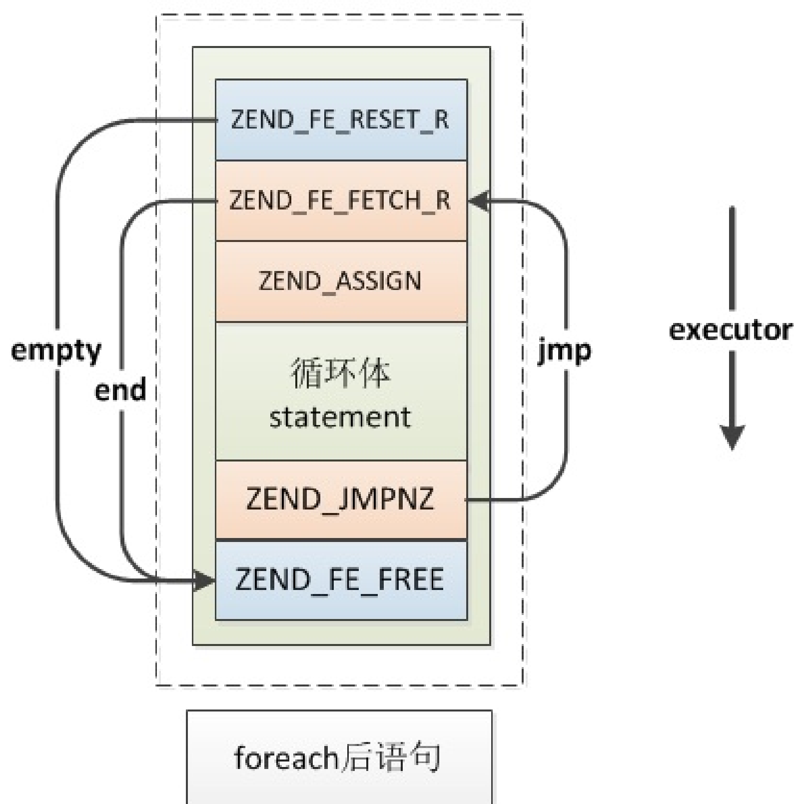
如果value是引用则在循环前首先将原数组或对象重置为引用类型，然后新分配一个zval指向这个引用，后面的过程就与上面的一致了，仍以上面的例子为例，如果是: `foreach($arr as $k=>&$v){ ... }` 则：



了解了foreach的实现、运行机制我们再回头看下其编译过程：

- **(1)** 编译拷贝数组、对象操作的指令：ZEND_FE_RESET_R，如果value是引用则是ZEND_FE_RESET_RW。执行时如果发现遍历的变量不是数组、对象，则抛出一个warning，然后跳出循环，所以这条指令还需要知道跳出的位置，这个位置需要编译完foreach以后才能确定；
- **(2)** 编译fetch数组/对象当前单元key、value的opcode： `ZEND_FE_FETCH_R` ，如果是引用则是 `ZEND_FE_FETCH_RW` ，此opcode还需要知道当遍历已经到达数组末尾时跳出遍历的位置，与步骤(1)的opcode相同,另外还有一个关键操作，前面已经说过遍历的key、value实际就是普通的局部变量，它们的内存存储位置正是在这一步分配确定的，分配过程与普通局部变量的过程完全相同，如果value不是一个CV变量(比如：foreach($arr as $v["xx"]){...})则还会编译其它操作的opcode；
- **(3)** 如果foreach定义了key则编译一条赋值opcode，此操作是对key进行赋值；
- **(4)** 编译循环体statement；
- **(5)** 编译跳回遍历开始位置的opcode： `ZEND_JMP` ，一次遍历结束时会跳回步骤(2)编译的opcode处进行下次遍历；
- **(6)** 设置步骤(1)、(2)两条opcode跳过的opcode数；
- **(7)** 编译 `ZEND_FE_FREE` ，此操作用于释放步骤(1)"拷贝"的数组。

最终编译后的结构：



运行时的步骤：

- **(1)** 执行 `ZEND_FE_RESET_R` ，过程上面已经介绍了；
- **(2)** 执行 `ZEND_FE_FETCH_R` ，此opcode的操作主要有三个：检查遍历位置是否到达末尾、将数组元素的value赋值给$value、将数组元素的key赋值给一个临时变量(注意与value不同)；
- **(3)** 如果定义了key则执行 `ZEND_ASSIGN` ，将key的值从临时变量赋值给$key，否则跳到步骤(4)；
- **(4)** 执行循环体的statement；
- **(5)** 执行 `ZEND_JMPNZ` 跳回步骤(2)；
- **(6)** 遍历结束后执行 `ZEND_FE_FREE` 释放数组。

PHP中还有几个与遍历相关的函数：

- current() - 返回数组中的当前单元
- each() - 返回数组中当前的键／值对并将数组指针向前移动一步
- end() - 将数组的内部指针指向最后一个单元
- next() - 将数组中的内部指针向前移动一位
- prev() - 将数组的内部指针倒回一位

4.4 中断及跳转

PHP中的中断及跳转语句主要有break、continue、goto，这几种语句的实现基础都是跳转。

4.4.1 break与continue

break用于结束当前for、foreach、while、do-while 或者 switch 结构的执行；continue用于跳过本次循环中剩余代码，进行下一轮循环。break、continue是非常相像的，它们都可以接受一个可选数字参数来决定跳过的循环层数，两者的不同点在于break是跳到循环结束的位置，而continue是跳到循环判断条件的位置，本质在于跳转位置的不同。

break、continue的实现稍微有些复杂，下面具体介绍下其编译过程。

上一节我们已经介绍过循环语句的编译，其中在各种循环编译过程中有两个特殊操作：zend_begin_loop()、zend_end_loop()，分别在循环编译前以及编译后调用，这两步操作就是为break、continue服务的。

在每层循环编译时都会创建一个 `zend_brk_cont_element` 的结构：

```
typedef struct _zend_brk_cont_element {
    int start;
    int cont;
    int brk;
    int parent;
} zend_brk_cont_element;
```

cont记录的是当前循环判断条件opcode起始位置，brk记录的是当前循环结束的位置，parent记录的是父层循环 `zend_brk_cont_element` 结构的存储位置，也就是说多层嵌套循环会生成一个 `zend_brk_cont_element` 的链表，每层循环编译结束时更新自己的 `zend_brk_cont_element` 结构，所以break、continue的处理过程实际就是根据跳出的层级索引到那一层的 `zend_brk_cont_element` 结构，然后得到它的cont、brk进行相应的opcode跳转。

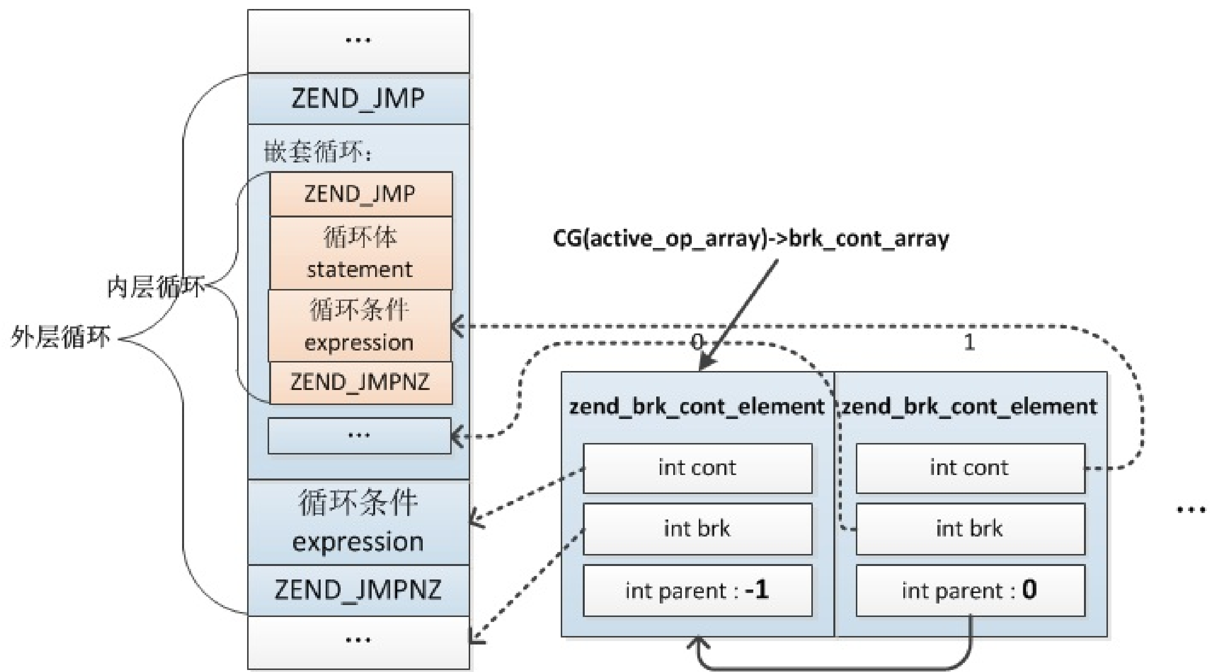
各循环的 `zend_brk_cont_element` 结构保存在 `zend_op_array->brk_cont_array` 数组中，编译各循环时依次申请一个 `zend_brk_cont_element` ，`zend_op_array->last_brk_cont` 记录此数组第一个可用位置，每申请一个元素last_brk_cont就相应的增加1，然后将数组扩容，parent记录的就是父层循环结构在该数组中的存储位置。

```
zend_brk_cont_element *get_next_brk_cont_element(zend_op_array *
op_array)
{
    op_array->last_brk_cont++;
    op_array->brk_cont_array = erealloc(op_array->brk_cont_array
, sizeof(zend_brk_cont_element)*op_array->last_brk_cont);
    return &op_array->brk_cont_array[op_array->last_brk_cont-1];
}
```

示例：

```
$i = 0;
while(1){
    while(1){
        if($i > 10){
            break 2;
        }
        ++$i
    }
}
```

循环编译完以后对应的内存结构：



介绍完编译循环结构时为break、continue做的准备，接下来我们具体分析下break、continue的编译。

有了前面的准备，break、continue的编译过程就比较简单了，主要就是各生成一条临时opcode：ZEND_BRK、ZEND_CONT，这条opcode记录着两个重要信息：

- **op1:** 记录着当前循环 `zend_brk_cont_element` 结构的存储位置(在循环编译过程中CG(context).current_brk_cont记录着当前循环zend_brk_cont_element的位置)
- **op2:** 记录着要跳出循环的层级，如果break/continue没有加数字，则默认为1

```
void zend_compile_break_continue(zend_ast *ast)
{
    zend_ast *depth_ast = ast->child[0];

    zend_op *opline;
    int depth;

    if (depth_ast) {
        zval *depth_zv;
        ...
        depth = Z_LVAL_P(depth_zv);
    } else {
        depth = 1;
    }
    ...

    //生成opcode
    opline = zend_emit_op(NULL, ast->kind == ZEND_AST_BREAK ? ZEND_BRK : ZEND_CONT, NULL, NULL);
    opline->op1.num = CG(context).current_brk_cont; //break、continue所在循环层
    opline->op2.num = depth;  //要跳出的层数
}
```

`zend_compile_break_continue()` 到这一步完成整个break、continue的编译还没有完成，因为 `CG(active_op_array)->brk_cont_array` 这个数组只是编译期间使用的一个临时结构，break、continue编译生成的opcode：ZEND*BRK*、*ZENDCONT*并不是运行时直接执行的，这条*opcode*在整个脚本编译完成后、执行前被优化为 __*ZEND_JMP*，这个操作在 `pass_two()` 中完成，关于这个过程在《3.1.2.2 AST->zend_op_array》一节曾经介绍过。

```
ZEND_API zend_op_array *compile_file(zend_file_handle *file_hand
le, int type)
{
    //语法解析
    zendparse();

    //AST->opcodes
    zend_compile_top_stmt(CG(ast));

    pass_two(op_array);
    ...
}
```

```
ZEND_API int pass_two(zend_op_array *op_array)
{
    ...

    opline = op_array->opcodes;
    end = opline + op_array->last;
    while (opline < end) {
        switch (opline->opcode) {
            ...
            case ZEND_BRK:
            case ZEND_CONT:
            {
                //计算跳转位置
                uint32_t jmp_target = zend_get_brk_cont_target(o
p_array, opline);
                ...
                //将opcode修改为ZEND_JMP
                opline->opcode = ZEND_JMP;
                opline->op1.opline_num = jmp_target;
                opline->op2.num = 0;

                //将绝对跳转opcode位置修改为相对当前opcode的位置
                ZEND_PASS_TWO_UPDATE_JMP_TARGET(op_array, opline
, opline->op1);
            }
            break;
            ...
        }
    }

    op_array->fn_flags |= ZEND_ACC_DONE_PASS_TWO;
    return 0;
}
```

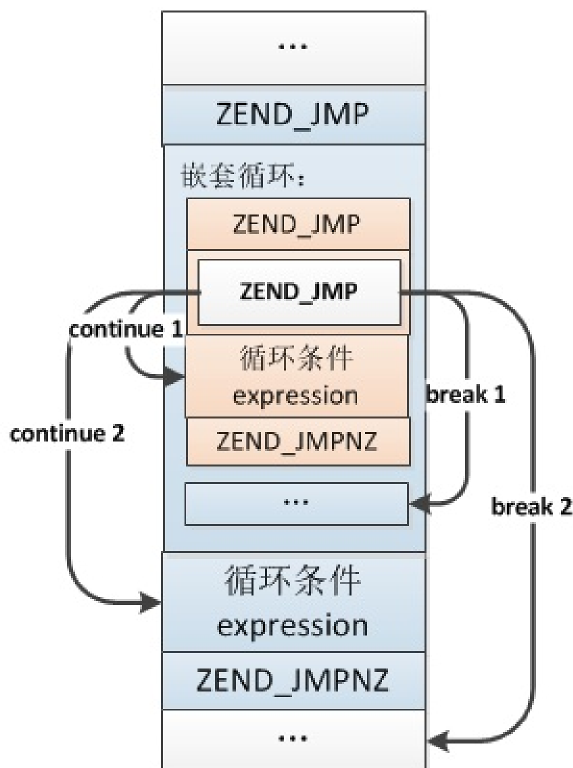
从上面的过程可以看出，如果opcode为：ZEND_BRK或ZEND_CONT则统一设置opcode为 `ZEND_JMP` ，新opcode的op1记录的是break、continue跳到opcode的位置，这个值根据编译期间的 `zend_brk_cont_element` 计算得到，首先从op1、op2取出break、continue所在循环的zend_brk_cont_element结构以及要跳过的层

级，然后根据 `zend_brk_cont_element.parent` 及层级数找到具体要跳出层的 `zend_brk_cont_element` 结构，从这个结构中获得那层循环判断条件及循环结束的opcode的位置。

```
static uint32_t zend_get_brk_cont_target(const zend_op_array *op
_array, const zend_op *opline) {
    int nest_levels = opline->op2.num; //跳出的层级：break n;
    int array_offset = opline->op1.num;//break、continue所属循环ze
nd_brk_cont_element的存储下标
    zend_brk_cont_element *jmp_to;
    do {
        //从break/continue所在循环层开始
        jmp_to = &op_array->brk_cont_array[array_offset];
        if (nest_levels > 1) {
            //如果还没到要跳出的层数则接着跳到上层
            array_offset = jmp_to->parent;
        }
    } while (--nest_levels > 0);

    return opline->opcode == ZEND_BRK ? jmp_to->brk : jmp_to->co
nt;
}
```

上面那个例子最终执行前的opcode如下图：



执行时直接跳到对应的opcode位置即可。

> **Note:**
>
> 在多层循环中break、continue直接根据层级数字跳转很不方便，这点PHP可以借鉴Golang的语法:break/continue + LABEL，支持按标签break、continue，根据上一节及本节介绍的内容这一个实现起来并不复杂，有兴趣的可以思考下如何实现。

4.4.2 goto

goto 操作符可以用来跳转到程序中的另一位置。该目标位置可以用目标名称加上冒号来标记，而跳转指令是 goto 之后接上目标位置的标记。PHP 中的 goto 有一定限制，目标位置只能位于同一个文件和作用域，也就是说无法跳出一个函数或类方法，也无法跳入到另一个函数，可以跳出循环但无法跳入循环(可以在同一层循环中跳转)，多层循环中通常会用goto代替多层break。

goto语法：

```
goto LABEL;

LABEL:
    statement;
```

goto与label需要组合使用，其实现与break、continue类似，最终也是被优化为 `ZEND_JMP` ，首先看下定义一个label时都有哪些操作：

```
statement:
    ...

    |   T_STRING ':' { $$ = zend_ast_create(ZEND_AST_LABEL, $1);
 }
;
```

label的编译过程非常简单，与循环结构的编译类似，编译时会把label插入 `CG(context).labels` 哈希表中，key就是label名称，value是一个 `zend_label` 结构：

```
typedef struct _zend_label {
    int brk_cont; //当前label所在循环
    uint32_t opline_num; //下一条opcode位置
} zend_label;
```

brk_cont用于记录当前label所在的循环，这个值就是上面介绍的每个循环在 `zend_op_array->brk_cont_array` 数组中的位置；opline_num比较容易理解，就是label下面第一条opcode的位置。到这里你应该能猜得到goto的工作过程了，首先根据label名称在 `CG(context).labels` 查找到跳转label的 `zend_label` 结构，然后jmp到 `zend_label.opline_num` 的位置，brk_cont的作用是用来判断是不是goto到了另一层循环中去。label具体的编译过程：

```
void zend_compile_label(zend_ast *ast)
{
    zend_string *label = zend_ast_get_str(ast->child[0]);
    zend_label dest;

    //编译时会将label插入CG(context).labels哈希表
    if (!CG(context).labels) {
        ALLOC_HASHTABLE(CG(context).labels);
        zend_hash_init(CG(context).labels, 8, NULL, label_ptr_dt
or, 0);
    }

    //设置label信息：当前所在循环、下一条opcode编号
    dest.brk_cont = CG(context).current_brk_cont;
    dest.opline_num = get_next_op_number(CG(active_op_array));

    if (!zend_hash_add_mem(CG(context).labels, label, &dest, siz
eof(zend_label))) {
        zend_error_noreturn(E_COMPILE_ERROR, "Label '%s' already
 defined", ZSTR_VAL(label));
    }
}
```

goto的编译过程：

```
void zend_compile_goto(zend_ast *ast)
{
    zend_ast *label_ast = ast->child[0];
    znode label_node;
    zend_op *opline;
    uint32_t opnum_start = get_next_op_number(CG(active_op_array
));

    zend_compile_expr(&label_node, label_ast);

    //如果当前在一个循环内则有的情况下是不能简单跳出循环的
    zend_handle_loops_and_finally();
    //编译一条临时opcode：ZEND_GOTO
    opline = zend_emit_op(NULL, ZEND_GOTO, NULL, &label_node);
    opline->op1.num = get_next_op_number(CG(active_op_array)) -
opnum_start - 1;
    opline->extended_value = CG(context).current_brk_cont;
}
```

goto初步被编译为 `ZEND_GOTO` ，其中label名称保存在op2，extended_value记录的是goto所在循环，如果没有在循环中这个值就等于-1，op1比较特殊，从上面编译的过程分析，它的值等于goto之间的opcode数，goto只编译了一条 `ZEND_GOTO` 哪来的其他opcode呢？这种情况就是goto在一个循环中，上一节介绍的循环结构中有一个比较特殊：foreach，它在遍历前会新生成一个zval用于遍历，这个zval是在循环结束时才被释放，假如foreach循环体中执行了goto，直接像普通跳转一样跳到了别的位置，那么这个zval就无法释放了，所以这种情况下在goto跳转前需要先执行这些收尾的opcode，这些opcode就是上面 `zend_handle_loops_and_finally()` 编译的，具体的细节这里不再展开，有兴趣的可以仔细研究下foreach编译时 `zend_begin_loop()` 的特殊处理。

后面的处理就与break、continue一样了，在 `pass_two()` 中 `ZEND_GOTO` 被重置为 `ZEND_JMP` ，具体的处理过程在 `zend_resolve_goto_label()` ，比较简单，不再赘述。

4.5 include/require

在实际应用中，我们不可能把所有的代码写到一个文件中，而是会按照一定的标准进行文件划分，include与require的功能就是将其他文件包含进来并且执行，比如在面向对象中通常会把一个类定义在单独文件中，使用时再include进来，类似其他语言中包的概念。

include与require没有本质上的区别，唯一的不同在于错误级别，当文件无法被正常加载时include会抛出warning警告，而require则会抛出error错误，本节下面的内容将以include说明。

在分析include的实现过程之前，首先要明确include的基本用法及特点：

- 被包含的文件将继承include所在行具有的全部变量范围，比如调用文件前面定义了一些变量，那么这些变量就能够在被包含的文件中使用，反之，被包含文件中定义的变量也将从include调用处开始可以被被调用文件所使用。
- 被包含文件中定义的函数、类在include执行之后将可以被随处使用，即具有全局作用域。
- include是在运行时加载文件并执行的，而不是在编译时。

这几个特性可以理解为include就是把其它文件的内容拷贝到了调用文件中，类似C语言中的宏(当然执行的时候并不是这样)，举个例子来说明：

```
//a.php
$var_1 = "hi";
$var_2 = array(1,2,3);

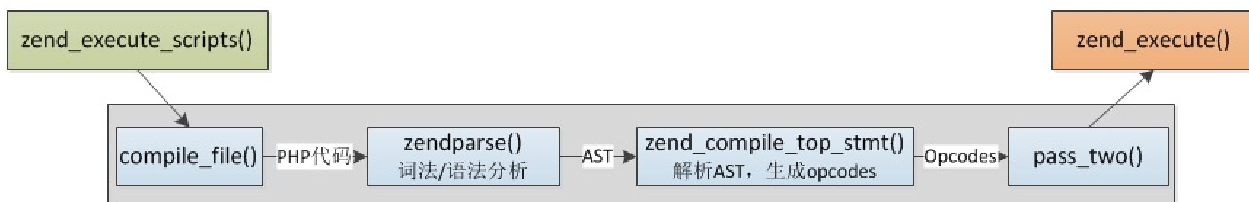
include 'b.php';

var_dump($var_2);
var_dump($var_3);

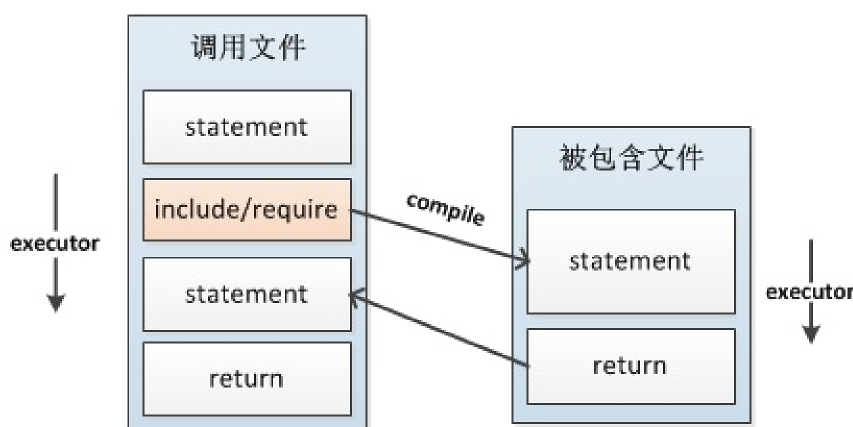
//b.php
$var_2 = array();
$var_3 = 9;
```

执行 `php a.php` 结果显示$var_2值被修改为array()了，而include文件中新定义的$var_3也可以在调用文件中使用。

接下来我们就以这个例子详细介绍下include具体是如何实现的。



前面我们曾介绍过Zend引擎的编译、执行两个阶段(见上图)，整个过程的输入是一个文件，然后经过 `PHP代码->AST->Opcodes->execute` 一系列过程完成整个处理，编译过程的输入是一个文件，输出是zend_op_array，输出接着成为执行过程的输入，而include的处理实际就是这个过程，执行include时把被包含的文件像主脚本一样编译然后执行，接着在回到调用处继续执行。



include的编译过程非常简单，只编译为一条opcode： `ZEND_INCLUDE_OR_EVAL` ，下面看下其具体处理过程：

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_INCLUDE_OR_EVA
L_SPEC_CONST_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    //include文件编译的zend_op_array
    zend_op_array *new_op_array=NULL;

    zval *inc_filename;
    zval tmp_inc_filename;
    zend_bool failure_retval=0;

    SAVE_OPLINE();
```

```
    inc_filename = EX_CONSTANT(opline->op1);
    ...

    switch (opline->extended_value) {
        ...
        case ZEND_INCLUDE:
        case ZEND_REQUIRE:
            //编译include的文件
            new_op_array = compile_filename(opline->extended_val
ue, inc_filename);
            break;
        ...
    }
    ...

    zend_execute_data *call;

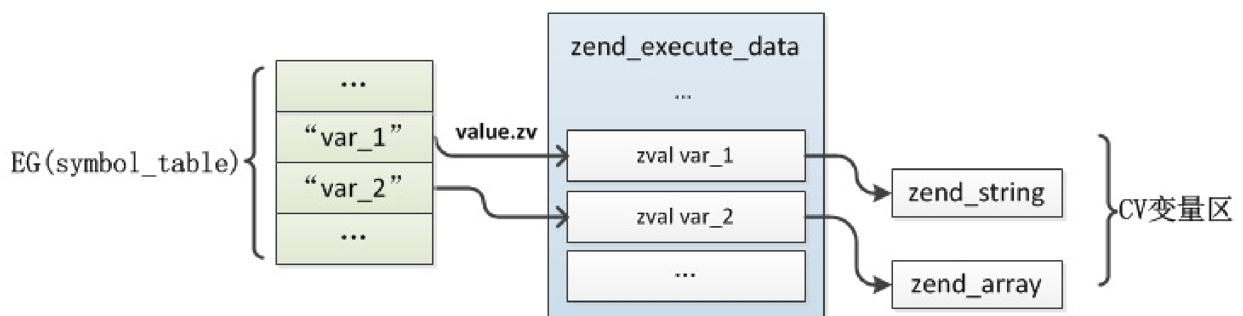
    //分配运行时的zend_execute_data
    call = zend_vm_stack_push_call_frame(ZEND_CALL_NESTED_CODE,
            (zend_function*)new_op_array, 0, EX(called_scope), Z
_OBJ(EX(This)));

    //继承调用文件的全局变量符号表
    if (EX(symbol_table)) {
        call->symbol_table = EX(symbol_table);
    } else {
        call->symbol_table = zend_rebuild_symbol_table();
    }
    //保存当前zend_execute_data，include执行完再还原
    call->prev_execute_data = execute_data;
    //执行前初始化
    i_init_code_execute_data(call, new_op_array, return_value);
    //zend_execute_ex执行器入口，如果没有自定义这个函数则默认为execute_
ex()
    if (EXPECTED(zend_execute_ex == execute_ex)) {
        //将执行器切到新的zend_execute_data，回忆下execute_ex()中的切
换过程
        ZEND_VM_ENTER();
    }
    ...
```

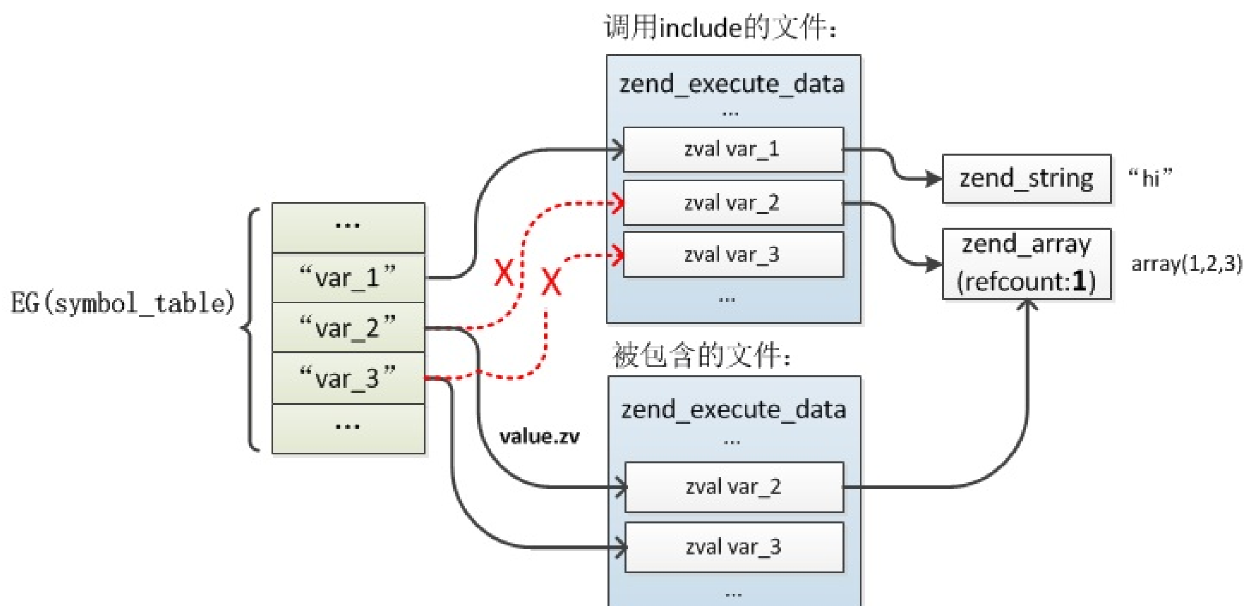
```
}
```

整个过程比较容易理解，编译的过程不再重复，与之前介绍的没有差别；执行的过程实际非常像函数的调用过程，首先也是重新分配了一个zend_execute_data，然后将执行器切到新的zend_execute_data，执行完以后再切回调用处，如果include文件中只定义了函数、类，没有定义全局变量则执行过程实际直接执行return，只是在编译阶段将函数、类注册到EG(function_table)、EG(class_table)中了，这种情况比较简单，但是如果有全局变量定义处理就比较复杂了，比如上面那个例子，两个文件中都定义了全局变量，这些变量是如何被继承、合并的呢？

上面的过程中还有一个关键操作： `i_init_code_execute_data()` ，关于这个函数在前面介绍 `zend_execute()` 时曾提过，这里面除了一些上下文的设置还会把当前zend_op_array下的变量移到EG(symbol_table)全局变量符号表中去，这些变量相对自己的作用域是局部变量，但它们定义在函数之外，实际也是全局变量，可以在函数中通过global访问，在执行前会把所有在php中定义的变量(zend_op_array->vars数组)插入EG(symbol_table)，value指向zend_execute_data局部变量的zval，如下图：

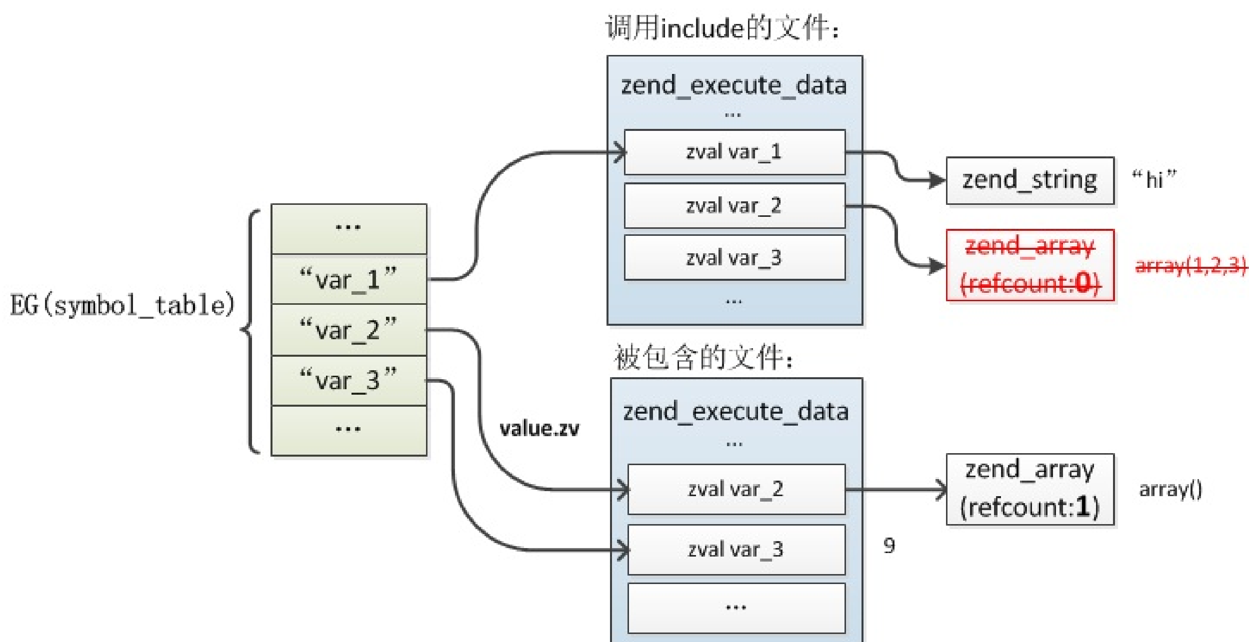


而include时也会执行这个步骤，如果发现var已经在EG(symbol_table)存在了，则会把value重新指向新的zval，也就是被包含文件的zend_execute_data的局部变量，同时会把原zval的value"拷贝"给新zval的value，概括一下就是被包含文件中的变量会继承、覆盖调用文件中的变量，这就是为什么被包含文件中可以直接使用调用文件中定义的变量的原因。被包含文件在 `zend_attach_symbol_table()` 完成以后EG(symbole_table)与zend_execute_data的关系：



> 注意：这里include文件中定义的var_2实际是替换了原文件中的变量，也就是只有一个var_2，所以此处zend_array的引用是1而不是2

接下来就是被包含文件的执行，执行到 `$var_2 = array()` 时，将原array(1,2,3)引用减1变为0，这时候将其释放，然后将新的value：array()赋给$var_2，这个过程就是普通变量的赋值过程，注意此时调用文件中的$var_2仍然指向被释放掉的value，此时的内存关系：



看到这里你可能会有一个疑问：$var_2既然被重新修改为新的一个值了，那么为什么调用文件中的$var_2仍然指向释放掉的value呢？include执行完成回到原来的调用文件中后为何可以读取到新的$var_2值以及新定义的var_3呢？答案在被包含文件执行完毕return的过程中。

被包含文件执行完以后最后执行return返回调用文件include的位置，return时会把被包含文件中的全局变量从zend_execute_data中移到EG(symbol_table)中，这里的移动是把value值更新到EG(symbol_table)，而不是像原来那样间接的指向value，这个操作在 `zend_detach_symbol_table()` 中完成，具体的return处理：

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL zend_leave_helper_S
PEC(ZEND_OPCODE_HANDLER_ARGS)
{
    ...
    if (EXPECTED((ZEND_CALL_KIND_EX(call_info) & ZEND_CALL_TOP)
== 0)) {
        //将include文件中定义的变量移到EG(symbol_table)
        zend_detach_symbol_table(execute_data);
        //释放zend_op_array
        destroy_op_array(&EX(func)->op_array);

        old_execute_data = execute_data;
        //切回调用文件的zend_execute_data
        execute_data = EG(current_execute_data) = EX(prev_execut
e_data);
        //释放include文件的zend_execute_data
        zend_vm_stack_free_call_frame_ex(call_info, old_execute_
data);

        //重新attach
        zend_attach_symbol_table(execute_data);

        LOAD_NEXT_OPLINE();
        ZEND_VM_LEAVE();
    }else{
        //函数、主脚本返回的情况
    }
}
```

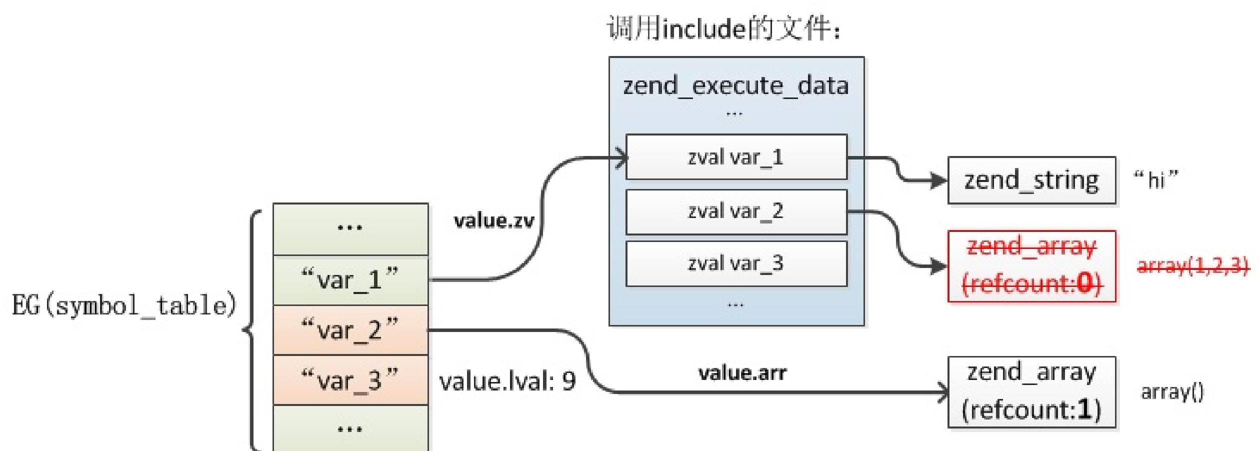
`zend_detach_symbol_table()` 操作：

```
ZEND_API void zend_detach_symbol_table(zend_execute_data *execut
e_data)
{
    zend_op_array *op_array = &execute_data->func->op_array;
    HashTable *ht = execute_data->symbol_table;

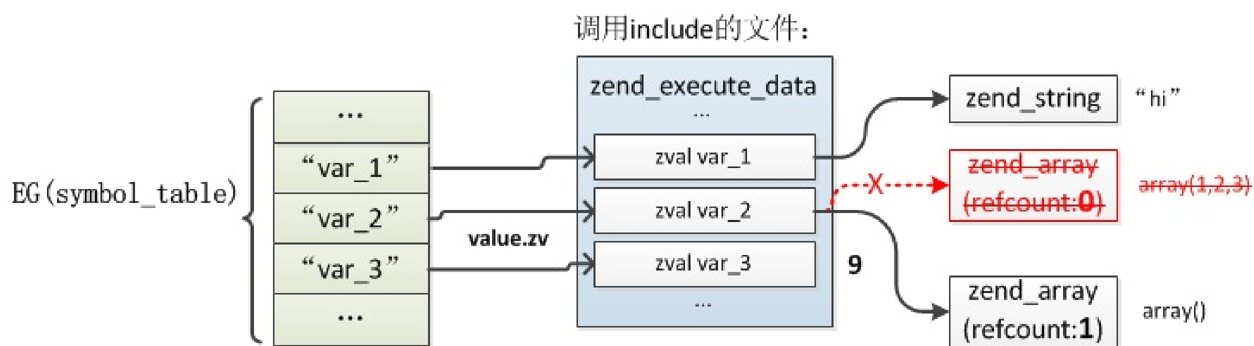
    /* copy real values from CV slots into symbol table */
    if (EXPECTED(op_array->last_var)) {
        zend_string **str = op_array->vars;
        zend_string **end = str + op_array->last_var;
        zval *var = EX_VAR_NUM(0);

        do {
            if (Z_TYPE_P(var) == IS_UNDEF) {
                zend_hash_del(ht, *str);
            } else {
                zend_hash_update(ht, *str, var);
                ZVAL_UNDEF(var);
            }
            str++;
            var++;
        } while (str != end);
    }
}
```

完成以后EG(symbol_table)：



接着是还原调用文件的zend_execute_data，切回调用文件的include位置，在将执行器切回之前再次执行了 `zend_attach_symbol_table()` ，这时就会将原调用文件的变量重新插入全局变量符号表，插入$var_2、$var_3时发现已经存在了，则将局部变量区的$var_2、$var_3的value修改为这个值，这就是$var_2被include文件更新后覆盖原value的过程，同时$var_3也因为在调用文件中出现了所以值被修改为include中设定的值，此时的内存关系：



这就是include的实现原理，整个过程并不复杂，比较难理解的一点在于两个文件之间变量的继承、覆盖，可以仔细研究下上面不同阶段时的内存关系图。

最后简单介绍下include_once、require_once，这两个与include、require的区别是在一次请求中同一文件只会被加载一次，第一次执行时会把这个文件保存在EG(included_files)哈希表中，再次加载时检查这个哈希表，如果发现已经加载过则直接跳过。

4.6 异常处理

PHP的异常处理与其它语言的类似，在程序中可以抛出、捕获一个异常，异常抛出必须只有定义在try{...}块中才可以被捕获，捕获以后将跳到catch块中进行处理，不再执行try中抛出异常之后的代码。

异常可以在任意位置抛出，然后将由最近的一个try所捕获，如果在当前执行空间没有进行捕获，那么将调用栈一直往上抛，比如在一个函数内部抛出一个异常，但是函数内没有进行try，而在函数调用的位置try了，那么就由调用处的catch捕获。

接下来我们从两个方面介绍下PHP异常处理的实现。

4.6.1 异常处理的编译

异常捕获及处理的语法：

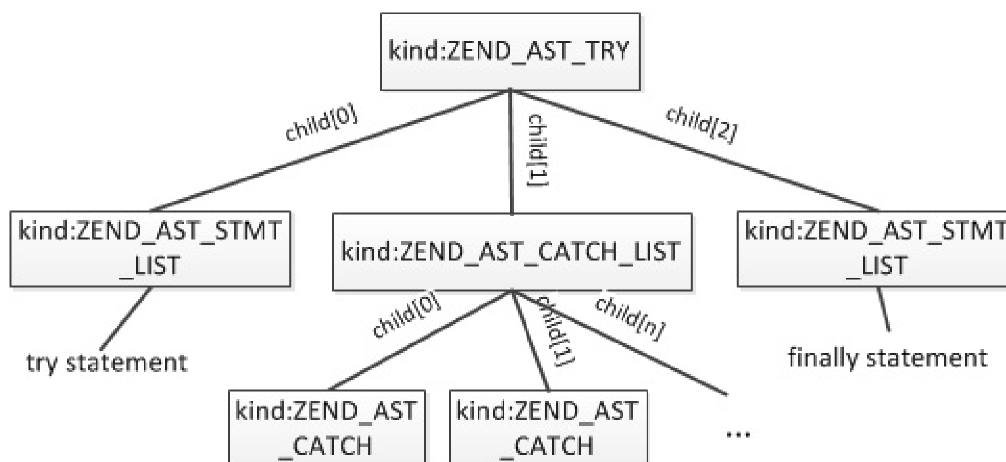
```
try{
    try statement;
}catch(exception_class_1 $e){
    catch statement 1;
}catch(exception_class_2 $e){
    catch statement 2;
}finally{
    finally statement;
}
```

try表示要捕获try statement中可能抛出的异常；catch是捕获到异常后的处理，可以定义多个，当try中抛出异常时会依次检查各个catch的异常类是否与抛出的匹配，如果匹配则有命中的那个catch块处理；finally为最后执行的代码，不管是否有异常抛出都会执行。

语法规则：

```
statement:
    ...
    |   T_TRY '{' inner_statement_list '}' catch_list finally_st
atement
            { $$ = zend_ast_create(ZEND_AST_TRY, $3, $5, $6); }
    ...
;
catch_list:
        /* empty */
            { $$ = zend_ast_create_list(0, ZEND_AST_CATCH_LIST);
 }
    |   catch_list T_CATCH '(' name T_VARIABLE ')' '{' inner_sta
tement_list '}'
            { $$ = zend_ast_list_add($1, zend_ast_create(ZEND_AS
T_CATCH, $4, $5, $8)); }
;
finally_statement:
        /* empty */ { $$ = NULL; }
    |   T_FINALLY '{' inner_statement_list '}' { $$ = $3; }
;
```

从语法规则可以看出，try-catch-finally最终编译为一个`ZEND_AST_TRY`节点，包含三个子节点，分别是：try statement、catch list、finally statement，try statement、finally statement就是普通的`ZEND_AST_STMT_LIST`节点，catch list包含多个`ZEND_AST_CATCH`节点，每个节点有三个子节点：exception class、exception object及catch statement，最终生成的AST：



具体的编译过程如下：

- **(1)** 向所属zend_op_array注册一个zend_try_catch_element结构，所有try都会注册一个这样的结构，与循环结构注册的zend_brk_cont_element类似，当前zend_op_array所有定义的异常保存在zend_op_array->try_catch_array数组中，这个结构用来记录try、catch以及finally开始的位置，具体结构：

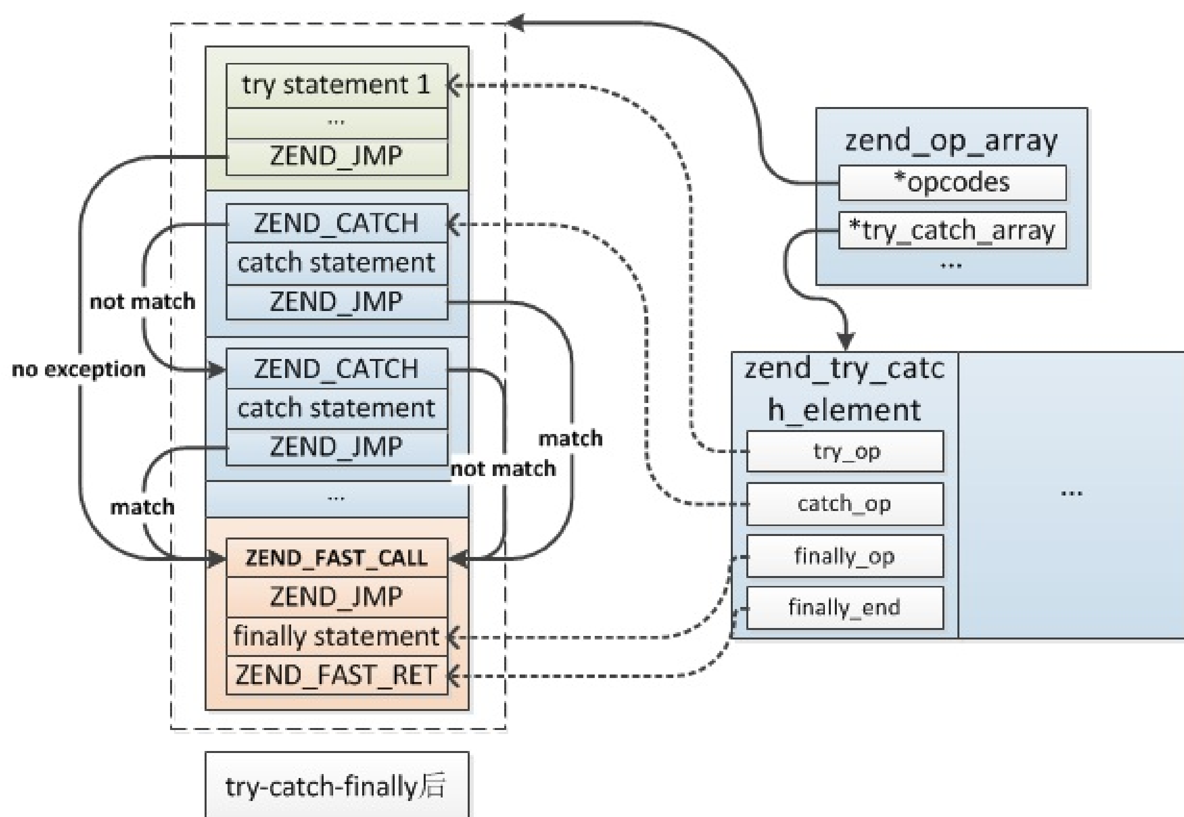
```
typedef struct _zend_try_catch_element {
  uint32_t try_op;     //try开始的opcode位置
  uint32_t catch_op;   //第1个catch块的opcode位置
  uint32_t finally_op; //finally开始的opcode位置
  uint32_t finally_end;//finally结束的opcode位置
} zend_try_catch_element;
```

- **(2)** 编译try statement，编译完以后如果定义了catch块则编译一条 `ZEND_JMP` ，此opcode的作用时当无异常抛出时跳过所有catch跳到finally或整个异常之外的，因为catch块是在try statement之后编译的，所以具体的跳转值目前还无法确定；

- **(3)** 依次编译各个catch块，如果没有定义则跳过此步骤，每个catch编译时首先编译一条 `ZEND_CATCH` ，此opcode保存着此catch的exception class、exception object以及下一个catch块开始的位置，编译第1个catch时将此opcode的位置记录在zend_try_catch_element.catch_op上，接着编译catch statement，最后编译一条 `ZEND_JMP` (最后一个catch不需要)，此opcode的作用与步骤(2)的相同；

- **(4)** 将步骤(2)、步骤(3)中 `ZEND_JMP` 跳转值设置为finally第1条opcode或异常定义之外的代码，如果没有定义finally则结束编译，否则编译finally块，首先编译一条 `ZEND_FAST_CALL` 及 `ZEND_JMP` ，接着编译finally statement，最后编译一条 `ZEND_FAST_RET` 。

编译完以后的结构：



异常的抛出通过throw一个异常对象来实现，这个对象必须继承>自Exception类，抛出异常的语法：

```
throw exception_object;
```

throw的编译比较简单，最终只编译为一条opcode： `ZEND_THROW` 。

4.6.2 异常的抛出与捕获

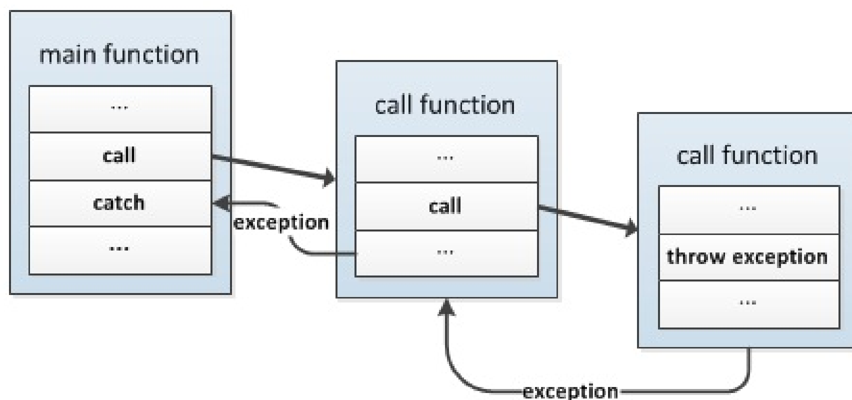
上一小节我们介绍了exception结构在编译阶段的处理，接下来我们再介绍下运行时exception的处理过程，这个过程相对比较复杂，整体的讲其处理流程整体如下：

- **(1)** 检查抛出的是否是object，否则将导致error错误；
- **(2)** 将EG(exception)设置为抛出的异常对象，同时将当前stack(即:zend_execute_data)接下来要执行的opcode设置为 `ZEND_HANDLE_EXCEPTION` ；
- **(3)** 执行 `ZEND_HANDLE_EXCEPTION` ，查找匹配的catch：
    - **(3.1)** 首先遍历当前zend_op_array下定义的所有异常捕获，即 `zend_op_array->try_catch_array` 数组，然后根据throw的位置、try开始的位置、catch开始的位置、finally开始的位置判断判断异常是否在

try范围内，如果同时命中了多个try(即嵌套try的情况)则选择最后那个(也就是最里层的)，遍历完以后如果命中了则进入步骤(3.2)处理，如果没有命中当前stack下任何try则进入步骤(4)；

  - **(3.2)** 到这一步表示抛出的异常在当前zend_op_array下有try拦截(注意这里只是表示异常在try中抛出的，但是抛出的异常并一定能被catch)，然后根据当前try块的 `zend_try_catch_element` 结构取出第一个catch的位置，将opcode设置为zend_try_catch_element.catch_op，跳到第一个catch块开始的位置执行，即:执行 `ZEND_CATCH` ；
  - **(3.3)** 执行 `ZEND_CATCH` ，检查抛出的异常对象是否与当前catch的类型匹配，检查的过程为判断两个类是否存在父子关系，如果匹配则表示异常被成功捕获，将EG(exception)清空，如果没有则跳到下一个catch的位置重复步骤(3.3)，如果到最后一个catch仍然没有命中则在这个catch的位置抛出一个异常(实际还是原来按个异常，只是将抛出的位置转移了当前catch的位置)，然后回到步骤(3);

- **(4)** 当前zend_op_array没能成功捕获异常，需要继续往上抛：回到调用位置，将接下来要执行的opcode设置为 `ZEND_HANDLE_EXCEPTION` ，比如函数中抛出了一个异常没有在函数中捕获，则跳到调用的位置继续捕获，回到步骤(3)；如果到最终主脚本也没有被捕获则将结束执行并导致error错误。



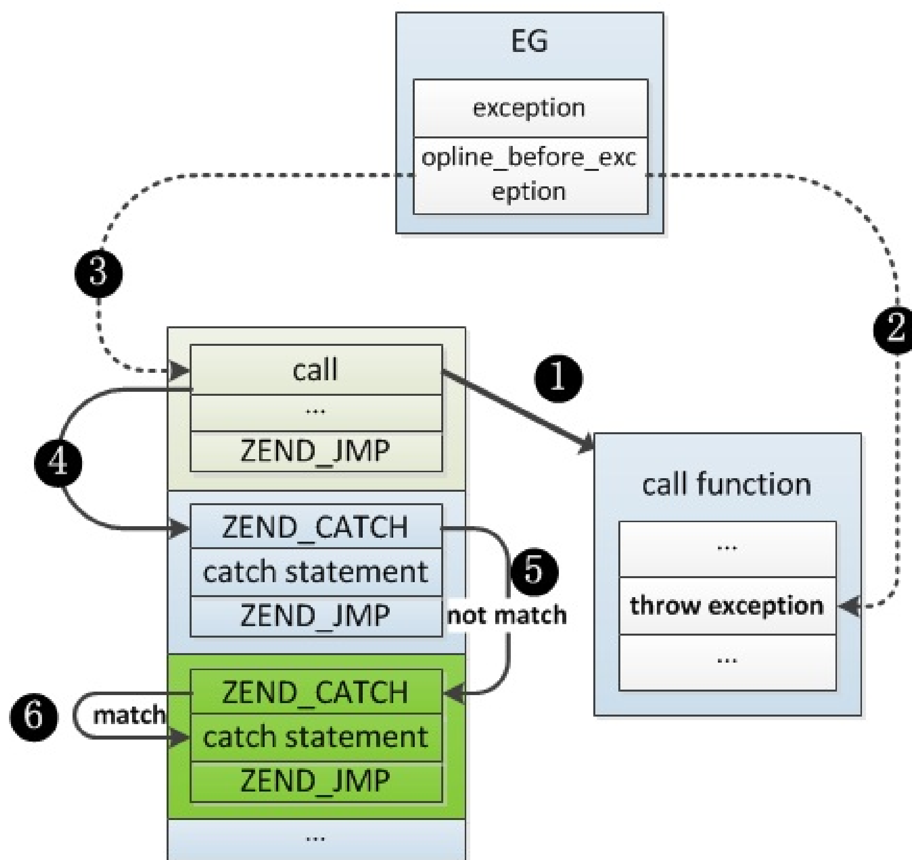
这个过程最复杂的地方在于异常匹配、传递的过程，主要为 `ZEND_HANDLE_EXCEPTION` 、 `ZEND_CATCH` 两条opcode之间的调用，当抛出一个异常时会终止后面opcode的执行，转向执行 `ZEND_HANDLE_EXCEPTION` ，根据异常抛出的位置定位到最近的一个try的catch位置，如果这个catch没有匹配则跳到下一个catch块，然后再次执行 `ZEND_HANDLE_EXCEPTION` ，如果到最后一个catch仍没有匹配则将异常抛出前位置EG(opline_before_exception)更新为最后一个catch的位置，再次执行 `ZEND_HANDLE_EXCEPTION` ，由于异常抛出的位置已经更新了所

以不会再匹配上次检查过的那个catch，这个过程实际就是不断递归执行 `ZEND_HANDLE_EXCEPTION` 、 `ZEND_CATCH` ；如果当前zend_op_array都无法捕获则将异常抛向上一个调用栈继续捕获，下面根据一个例子具体说明下：

```
function my_func(){
    //...
    throw new Exception("This is a exception from my_func()");
}

try{
    my_func();
}catch(ErrorException $e){
    echo "ErrorException";
}catch(Exception $e){
    echo "Exception";
}
```

my_func()中抛出了一个异常，首先在my_func()中抛出一个异常，然后在my_func()的zend_op_array中检查是不是能够捕获，发现没有，则回到调用的位置，再次检查，第1次匹配到 `catch(ErrorException $e)` ，检查后发现并不匹配，然后跳到下一个catch块继续匹配，第2次匹配到 `catch(Exception $e)` ，检查后发现命中，捕获成功。



上面的过程并没有提到finally的执行时机，首先要明确finally在哪些情况下会执行，命中catch的情况比较简单，即在catch statement执行完以后跳到finally执行，另外一种情况是如果一个异常在try中但没有命中任何catch那么其finally也是会被执行的，这种情况的finally实际是在步骤(3)中执行的，最后一个catch检查完以后会更新异常抛出位置：EG(opline_before_exception)，然后会再次执行 `ZEND_HANDLE_EXCEPTION` ，再次检查时就会发现没有命中任何catch但命中finally了(因为异常位置更新了)，这时候就会将异常对象保存在finally块中，然后执行finally，执行完再将异常对象还原继续捕获，下面看下步骤(3)的具体处理过程：

```
static ZEND_OPCODE_HANDLER_RET ZEND_FASTCALL ZEND_HANDLE_EXCEPTI
ON_SPEC_HANDLER(ZEND_OPCODE_HANDLER_ARGS)
{
    //op_num为异常抛出的位置，根据异常抛出前最后一条opcode与第一条opcode
计算得出
    uint32_t op_num = EG(opline_before_exception) - EX(func)->op
_array.opcodes;

    uint32_t catch_op_num = 0, finally_op_num = 0, finally_op_en
d = 0;
```

```
    //查找异常是不是被try了：找最近的一层try
    for (i = 0; i < EX(func)->op_array.last_try_catch; i++) {
        if (EX(func)->op_array.try_catch_array[i].try_op > op_nu
m) {
            //try在抛出之后
            break;
        }
        in_finally = 0;
        //异常抛出位置在try后且比第一个catch位置小，表明这个try有可能捕获
异常
        if (op_num < EX(func)->op_array.try_catch_array[i].catch
_op) {
            //第一个catch的位置
            catch_op_num = EX(func)->op_array.try_catch_array[i]
.catch_op;
        }
        //当前try有finally
        if (op_num < EX(func)->op_array.try_catch_array[i].final
ly_op) {
            finally_op_num = EX(func)->op_array.try_catch_array[
i].finally_op;
            finally_op_end = EX(func)->op_array.try_catch_array[
i].finally_end;
        }
        if (op_num >= EX(func)->op_array.try_catch_array[i].fina
lly_op &&
                op_num < EX(func)->op_array.try_catch_array[i].f
inally_end) {
            finally_op_end = EX(func)->op_array.try_catch_array[
i].finally_end;
            in_finally = 1;
        }
    }

    cleanup_unfinished_calls(execute_data, op_num);

    //异常命中了try但没有命中任何catch且那个try定义了finally：需要执行fi
nally
    //catch_op_num >= finally_op_num是嵌套try的情况，因为finally是检
查完所有catch、更新异常抛出位置之后再执行的
```

```
    //所以检查完内层try再检查外层循环时会出现这种情况
    if (finally_op_num && (!catch_op_num || catch_op_num >= finally_op_num)) {
        zval *fast_call = EX_VAR(EX(func)->op_array.opcodes[finally_op_end].op1.var);

        cleanup_live_vars(execute_data, op_num, finally_op_num);
        if (in_finally && Z_OBJ_P(fast_call)) {
            zend_exception_set_previous(EG(exception), Z_OBJ_P(fast_call));
        }
        //临时将EG(exception)转移到finally下，执行完finally再抛出
        Z_OBJ_P(fast_call) = EG(exception);
        EG(exception) = NULL;
        fast_call->u2.lineno = (uint32_t)-1;
        ZEND_VM_SET_OPCODE(&EX(func)->op_array.opcodes[finally_op_num]);
        ZEND_VM_CONTINUE();
    }else{
        //这个是善后处理，因为异常抛出后后面的opcode将不再执行，但有些情况下还需要把一些资源释放掉
        //比如前面我们介绍goto时提到的foreach中是不能直接跳出的，throw也是类似
        cleanup_live_vars(execute_data, op_num, catch_op_num);
        ...
        if (catch_op_num) {
            //匹配到catch(但不一定命中)，跳到catch处执行ZEND_CATCH进行判断
            ZEND_VM_SET_OPCODE(&EX(func)->op_array.opcodes[catch_op_num]);
            ZEND_VM_CONTINUE();
        } else if (UNEXPECTED((EX(func)->op_array.fn_flags & ZEND_ACC_GENERATOR) != 0)) {
            ...
        } else {
            //当前zend_op_array下已经没有匹配到的try了，如果异常仍没有被捕获则将在zend_leave_helper_SPEC()将异常抛给prev_execute_data继续捕获
            ZEND_VM_TAIL_CALL(zend_leave_helper_SPEC(ZEND_OPCODE_HANDLER_ARGS_PASSTHRU));
```

```
            }
        }
    }
```

具体的实现过程还有很多额外的处理，这里不再展开，感兴趣的可以详细研究下 `ZEND_HANDLE_EXCEPTION` 、 `ZEND_CATCH` 两条opcode以及zend_exception.c中具体逻辑。

4.6.3 内核的异常处理

前面介绍的异常处理是PHP语言层面的实现，在内核中也有一套供内核使用的异常处理模型，也就是C语言异常处理的实现，如：

```
static int php_start_sapi(void)
{
    ...

    zend_try {
        ...
    } zend_catch {
        ...
    } zend_end_try();
    ...
}
```

C语言并没有在语言层面提供try-catch机制，那么PHP中的是如何实现的呢？这个主要利用sigsetjmp()、siglongjmp()两个函数实现堆栈的保存、还原，在try的位置通过sigsetjmp()将当前位置的堆栈保存在一个变量中，异常抛出通过siglongjmp()跳回原位置，具体看下这几个宏的定义：

```
#define zend_try
\
    {
\
        JMP_BUF *__orig_bailout = EG(bailout);
\
        JMP_BUF __bailout;
\

\
        EG(bailout) = &__bailout;
\
        if (SETJMP(__bailout)==0) {
#define zend_catch
\
        } else {
\
            EG(bailout) = __orig_bailout;
#define zend_end_try()
\
        }
\
        EG(bailout) = __orig_bailout;
\
    }


# define JMP_BUF sigjmp_buf
# define SETJMP(a) sigsetjmp(a, 0)
# define LONGJMP(a,b) siglongjmp(a, b)
# define JMP_BUF sigjmp_buf
```

展开后：

```
{
    //保存上一个zend_try记录的JMP_BUF，目的是实现多层嵌套try
    JMP_BUF *__orig_bailout = EG(bailout);
    JMP_BUF __bailout;

    //将当前堆栈保存在__bailout
    EG(bailout) = &__bailout;
    if (SETJMP(__bailout)==0) {
        //try中的代码
        //抛出异常调用：LONGJMP()
    }else { //异常抛出后到这个分支
        EG(bailout) = __orig_bailout;
    }
    EG(bailout) = __orig_bailout;
}
```

5.1 Zend内存池

zend针对内存的操作封装了一层，用于替换直接的内存操作：malloc、free等，实现了更高效率的内存利用，其实现主要参考了tcmalloc的设计。

源码中emalloc、efree、estrdup等等就是内存池的操作。

内存池是内核中最底层的内存操作，定义了三种粒度的内存块：chunk、page、slot，每个chunk的大小为2M，page大小为4KB，一个chunk被切割为512个page，而一个或若干个page被切割为多个slot，所以申请内存时按照不同的申请大小决定具体的分配策略：

- **Huge(chunk):** 申请内存大于2M，直接调用系统分配，分配若干个chunk
- **Large(page):** 申请内存大于3092B(3/4 page_size)，小于2044KB(511 page_size)，分配若干个page
- **Small(slot):** 申请内存小于等于3092B(3/4 page_size)，内存池提前定义好了30种同等大小的内存(8,16,24,32，...3072)，他们分配在不同的page上(不同大小的内存可能会分配在多个连续的page)，申请内存时直接在对应page上查找可用位置

5.1.1 基本数据结构

chunk由512个page组成，其中第一个page用于保存chunk结构，剩下的511个page用于内存分配，page主要用于Large、Small两种内存的分配；heap是表示内存池的一个结构，它是最主要的一个结构，用于管理上面三种内存的分配，Zend中只有一个heap结构。

```
struct _zend_mm_heap {
#if ZEND_MM_STAT
    size_t              size; //当前已用内存数
    size_t              peak; //内存单次申请的峰值
#endif
    zend_mm_free_slot *free_slot[ZEND_MM_BINS]; // 小内存分配的可用位置链表，ZEND_MM_BINS等于30，即此数组表示的是各种大小内存对应的链表头部
    ...

    zend_mm_huge_list *huge_list;               //大内存链表
```

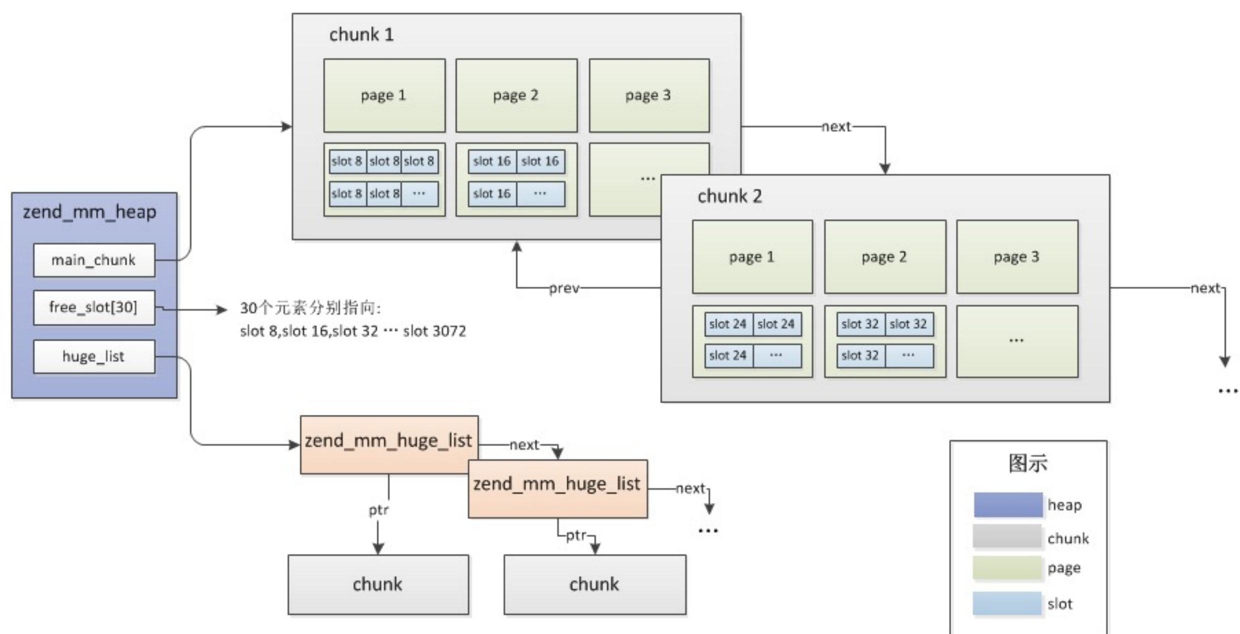
```
    zend_mm_chunk     *main_chunk;              //指向chunk链表头部

    zend_mm_chunk     *cached_chunks;           //缓存的chunk链表
    int                chunks_count;            //已分配chunk数
    int                peak_chunks_count;       //当前request使用chunk峰值
    int                cached_chunks_count;     //缓存的chunk数
    double             avg_chunks_count;        //chunk使用均值，每次请求结束后会根据peak_chunks_count重新计算：(avg_chunks_count+peak_chunks_count)/2.0
}

struct _zend_mm_chunk {
    zend_mm_heap      *heap; //指向heap
    zend_mm_chunk     *next; //指向下一个chunk
    zend_mm_chunk     *prev; //指向上一个chunk
    int                free_pages; //当前chunk的剩余page数
    int                free_tail;               /* number of free pages at the end of chunk */
    int                num;
    char               reserve[64 - (sizeof(void*) * 3 + sizeof(int) * 3)];
    zend_mm_heap       heap_slot; //heap结构，只有主chunk会用到
    zend_mm_page_map   free_map; //标识各page是否已分配的bitmap数组，总大小512bit，对应page总数，每个page占一个bit位
    zend_mm_page_info  map[ZEND_MM_PAGES]; //各page的信息：当前page使用类型(用于large分配还是small)、占用的page数等
};

//按固定大小切好的small内存槽
struct _zend_mm_free_slot {
    zend_mm_free_slot *next_free_slot;//此指针只有内存未分配时用到，分配后整个结构体转为char使用
};
```

chunk、page、slot三者的关系：



接下来看下内存池的初始化以及三种内存分配的过程。

5.1.2 内存池初始化

内存池在php_module_startup阶段初始化，start_memory_manager()：

```
ZEND_API void start_memory_manager(void)
{
#ifdef ZTS
    ts_allocate_id(&alloc_globals_id, sizeof(zend_alloc_globals)
, (ts_allocate_ctor) alloc_globals_ctor, (ts_allocate_dtor) allo
c_globals_dtor);
#else
    alloc_globals_ctor(&alloc_globals);
#endif
}

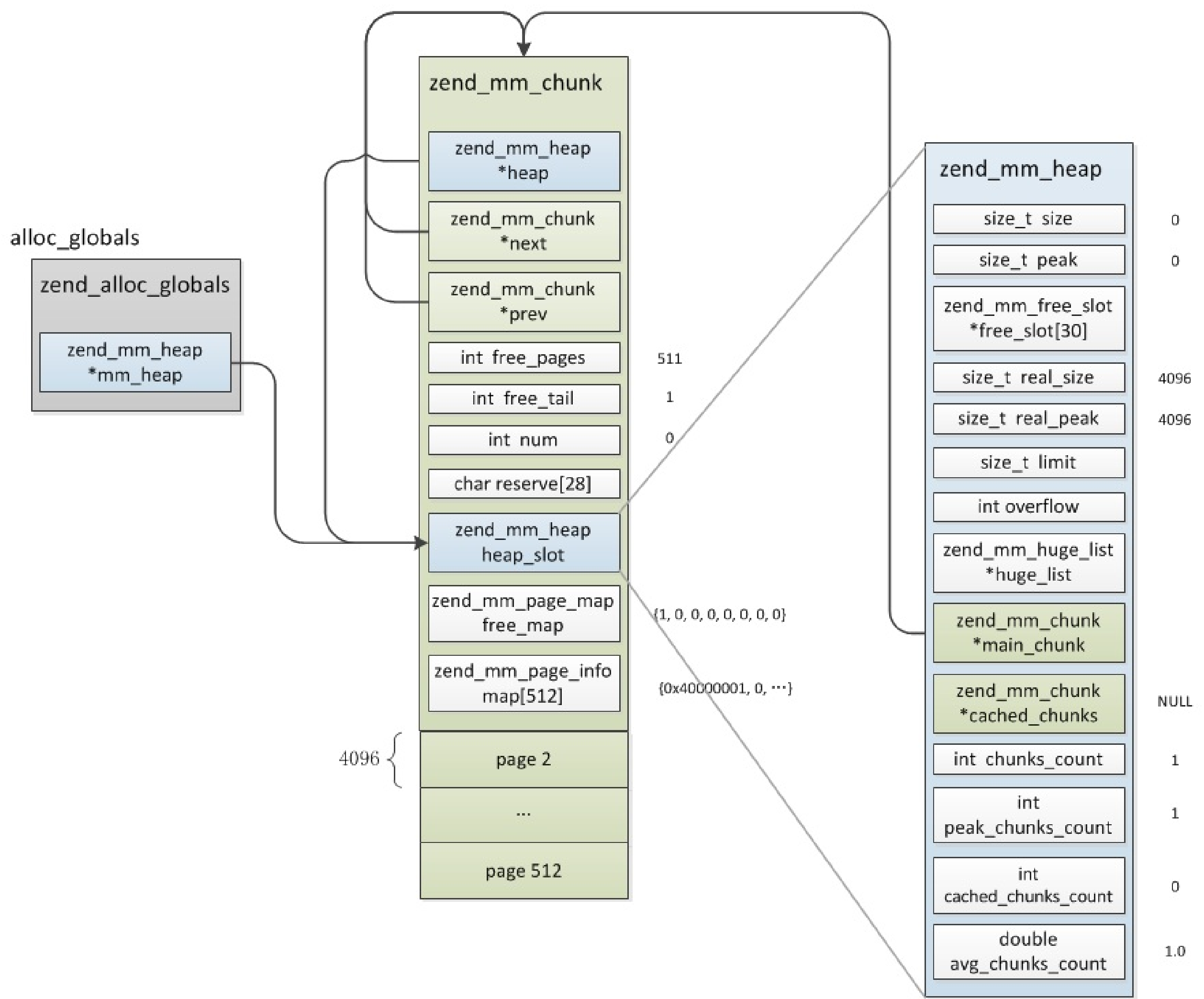
static void alloc_globals_ctor(zend_alloc_globals *alloc_globals
)
{
#ifdef MAP_HUGETLB
    tmp = getenv("USE_ZEND_ALLOC_HUGE_PAGES");
    if (tmp && zend_atoi(tmp, 0)) {
        zend_mm_use_huge_pages = 1;
    }
#endif
    ZEND_TSRMLS_CACHE_UPDATE();
    alloc_globals->mm_heap = zend_mm_init();
}
```

alloc_globals 是一个全局变量，即 **AG**宏 ，它只有一个成员:mm_heap，保存着整个内存池的信息，所有内存的分配都是基于这个值，多线程模式下(ZTS)会有多个heap，也就是说每个线程都有一个独立的内存池，看下它的初始化：

```
static zend_mm_heap *zend_mm_init(void)
{
    //向系统申请2M大小的chunk
    zend_mm_chunk *chunk = (zend_mm_chunk*)zend_mm_chunk_alloc_i
nt(ZEND_MM_CHUNK_SIZE, ZEND_MM_CHUNK_SIZE);
    zend_mm_heap *heap;

    heap = &chunk->heap_slot; //heap结构实际是主chunk嵌入的一个结构，
后面再分配chunk的heap_slot不再使用
    chunk->heap = heap;
    chunk->next = chunk;
    chunk->prev = chunk;
    chunk->free_pages = ZEND_MM_PAGES - ZEND_MM_FIRST_PAGE; //剩
余可用page数
    chunk->free_tail = ZEND_MM_FIRST_PAGE;
    chunk->num = 0;
    chunk->free_map[0] = (Z_L(1) << ZEND_MM_FIRST_PAGE) - 1; //
将第一个page的bit分配标识位设置为1
    chunk->map[0] = ZEND_MM_LRUN(ZEND_MM_FIRST_PAGE); //第一个pag
e的类型为ZEND_MM_IS_LRUN，即large内存
    heap->main_chunk = chunk; //指向主chunk
    heap->cached_chunks = NULL; //缓存chunk链表
    heap->chunks_count = 1; //已分配chunk数
    heap->peak_chunks_count = 1;
    heap->cached_chunks_count = 0;
    heap->avg_chunks_count = 1.0;
    ...
    heap->huge_list = NULL; //huge内存链表
    return heap;
}
```

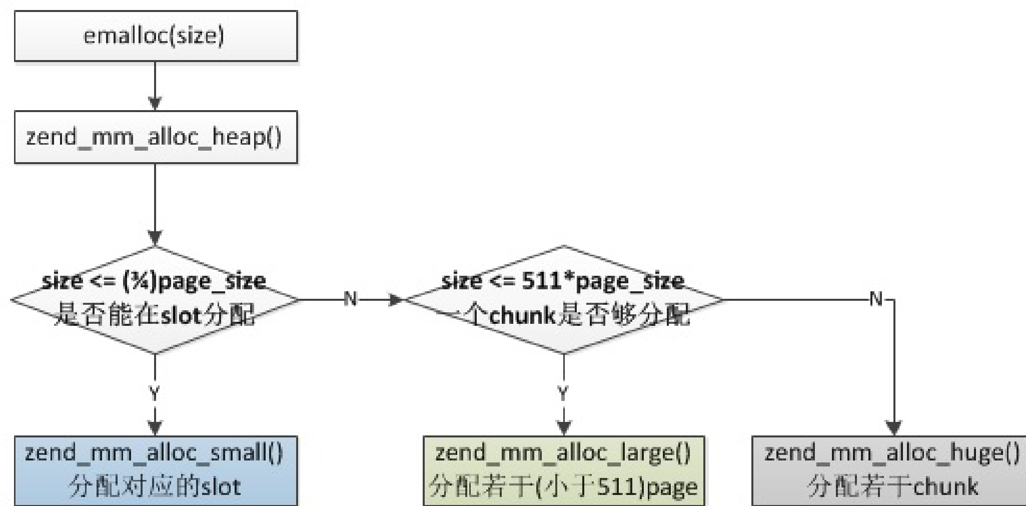
这里分配了主chunk，只有第一个chunk的heap会用到，后面分配的chunk不再用到heap，初始化完的结构如下图：



初始化的过程实际只是分配了一个主chunk，这里并没有看到开始提到的小内存slot切割，下一节我们来详细看下各种内存的分配过程。

5.1.3 内存分配

文章开头已经简单提过Zend内存分配器按照申请内存的大小有三种不同的实现：



5.1.3.1 Huge分配

超过2M内存的申请，与通用的内存申请没有太大差别，只是将申请的内存块通过单链表进行了管理。

```
static void *zend_mm_alloc_huge(zend_mm_heap *heap, size_t size
ZEND_FILE_LINE_DC ZEND_FILE_LINE_ORIG_DC)
{
    size_t new_size = ZEND_MM_ALIGNED_SIZE_EX(size, REAL_PAGE_SI
ZE); //按页大小重置实际要分配的内存

#if ZEND_MM_LIMIT
    //如果有内存使用限制则check是否已达上限，达到的话进行zend_mm_gc清理后
再检查
    //此过程不再展开分析
#endif

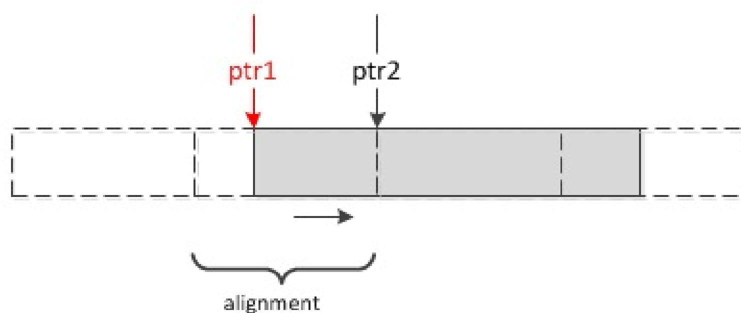
    //分配chunk
    ptr = zend_mm_chunk_alloc(heap, new_size, ZEND_MM_CHUNK_SIZE
);
    if (UNEXPECTED(ptr == NULL)) {
        //清理后再尝试分配一次
        if (zend_mm_gc(heap) &&
            (ptr = zend_mm_chunk_alloc(heap, new_size, ZEND_MM_C
HUNK_SIZE)) != NULL) {
            /* pass */
        } else {
            //申请失败
            zend_mm_safe_error(heap, "Out of memory");
            return NULL;
        }
    }

    //将申请的内存通过zend_mm_huge_list插入到链表中,heap->huge_list指
向的实际是zend_mm_huge_list
    zend_mm_add_huge_block(heap, ptr, new_size, ...);
    ...

    return ptr;
}
```

huge的分配实际就是分配多个chunk，chunk的分配也是large、small内存分配的基础，它是ZendMM向系统申请内存的唯一粒度。在申请chunk内存时有一个关键操作，那就是将内存地址对齐到ZEND_MM_CHUNK_SIZE，也就是说申请的chunk

地址都是ZEND_MM_CHUNK_SIZE的整数倍，注意：这里说的内存对齐值并不是系统的字节对齐值，所以需要在申请后自己调整下。ZendMM的处理方法是：先按实际要申请的内存大小申请一次，如果系统分配的地址恰好是ZEND_MM_CHUNK_SIZE的整数倍那么就不需要调整了，直接返回使用；如果不是ZEND_MM_CHUNK_SIZE的整数倍，ZendMM会把这块内存释放掉，然后按照"实际要申请的内存大小+ZEND_MM_CHUNK_SIZE"的大小重新申请一块内存，多申请的ZEND_MM_CHUNK_SIZE大小的内存是用来调整的，ZendMM会从系统分配的地址向后偏移到ZEND_MM_CHUNK_SIZE的整数倍位置，调整完以后会把多余的内存再释放掉，如下图所示,虚线部分为alignment大小的内容，灰色部分为申请的内容大小，系统返回的地址为ptr1，而实际使用的内存是从ptr2开始的。



下面看下chunk的具体分配过程：

```
//size为申请内存的大小，alignment为内存对齐值，一般为ZEND_MM_CHUNK_SIZE

static void *zend_mm_chunk_alloc_int(size_t size, size_t alignme
nt)
{
    //向系统申请size大小的内存
    void *ptr = zend_mm_mmap(size);
    if (ptr == NULL) {
        return NULL;
    } else if (ZEND_MM_ALIGNED_OFFSET(ptr, alignment) == 0) {//
判断申请的内存是否为alignment的整数倍
        //是的话直接返回
        return ptr;
    }else{
        //申请的内存不是按照alignment对齐的，注意这里的alignment并不是
系统的字节对齐值
        size_t offset;

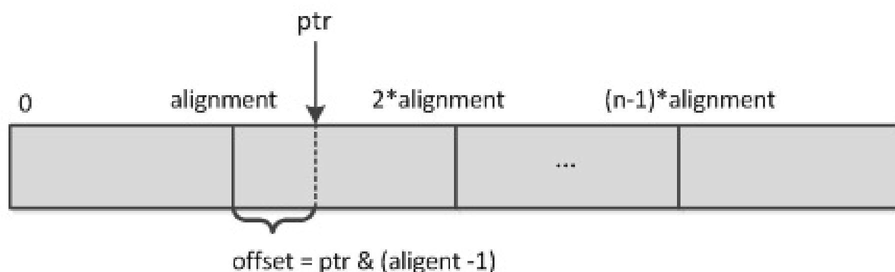
        //将申请的内存释放掉重新申请
```

```
        zend_mm_munmap(ptr, size);
        //重新申请一块内存，这里会多申请一块内存，用于截取到alignment的整数倍，可以忽略REAL_PAGE_SIZE
        ptr = zend_mm_mmap(size + alignment - REAL_PAGE_SIZE);
        //offset为ptr距离上一个alignment对齐内存位置的大小，注意不能往前移，因为前面的内存都是分配了的
        offset = ZEND_MM_ALIGNED_OFFSET(ptr, alignment);
        if (offset != 0) {
            offset = alignment - offset;
            zend_mm_munmap(ptr, offset);
            //偏移ptr，对齐到alignment
            ptr = (char*)ptr + offset;
            alignment -= offset;
        }
        if (alignment > REAL_PAGE_SIZE) {
            zend_mm_munmap((char*)ptr + size, alignment - REAL_PAGE_SIZE);
        }
        return ptr;
    }
}
```

这个过程中用到了一个宏：

```
#define ZEND_MM_ALIGNED_OFFSET(size, alignment) \
    (((size_t)(size)) & ((alignment) - 1))
```

这个宏的作用是计算按alignment对齐的内存地址距离上一个alignment整数倍内存地址的大小，alignment必须为2的n次方，比如一段n*alignment大小的内存，ptr为其中一个位置，那么就可以通过位运算计算得到ptr所属内存块的offset：



这个位运算是因为alignment为2^n，所以可以通过alignment取到最低位的位置，也就是相对上一个整数倍alignment的offset，实际如果不用运算的话可以通过：`offset = (ptr/alignment取整)*alignment - ptr` 得到，这个更容易理解些。

5.1.3.2 Large分配

大于3/4的pagesize(4KB)且小于等于511个pagesize的内存申请，也就是一个chunk的大小够用(之所以是511个page而不是512个是因为第一个page始终被chunk结构占用)，__如果申请多个page的话 分配的时候这些page都是连续的 。

```
static zend_always_inline void *zend_mm_alloc_large(zend_mm_heap
 *heap, size_t size ZEND_FILE_LINE_DC ZEND_FILE_LINE_ORIG_DC)
{
    //根据size大小计算需要分配多少个page
    int pages_count = (int)ZEND_MM_SIZE_TO_NUM(size, ZEND_MM_PAG
E_SIZE);

    //分配pages_count个page
    void *ptr = zend_mm_alloc_pages(heap, pages_count, ...);

    ...

    return ptr;
}
```

进一步看下 `zend_mm_alloc_pages` ，这个过程比较复杂，简单描述的话就是从第一个chunk开始查找当前chunk下是否有pagescount个连续可用的page，有的话就停止查找，没有的话则接着查找下一个chunk，如果直到最后一个chunk也没找到则重新分配一个新的chunk并插入chunk链表，这个过程中最不好理解的一点在于如何查找pagescount个连续可用的page，这个主要根据 ***chunk->free_map*** 实现的，在看具体执行过程之前我们先解释下 __free_map 的作用:

我们已经知道每个**chunk**由**512**个**page**组成，而不管是**large**分配还是**small**分配，其分配的最小粒子都是**page(small**也是先分配**1**个或多个**page**然后再进行的切割**)**，所以需要有一个数组来记录每个**page**是否已经分配，**free_map**的作用就是标识当前**chunk**下各**page**的分配与否，比较特别的是**free_map**并不是**512**大小的数组，因为需要记录的信息非常简单，只需要一个**bit**位就够了，所以**free_map**就

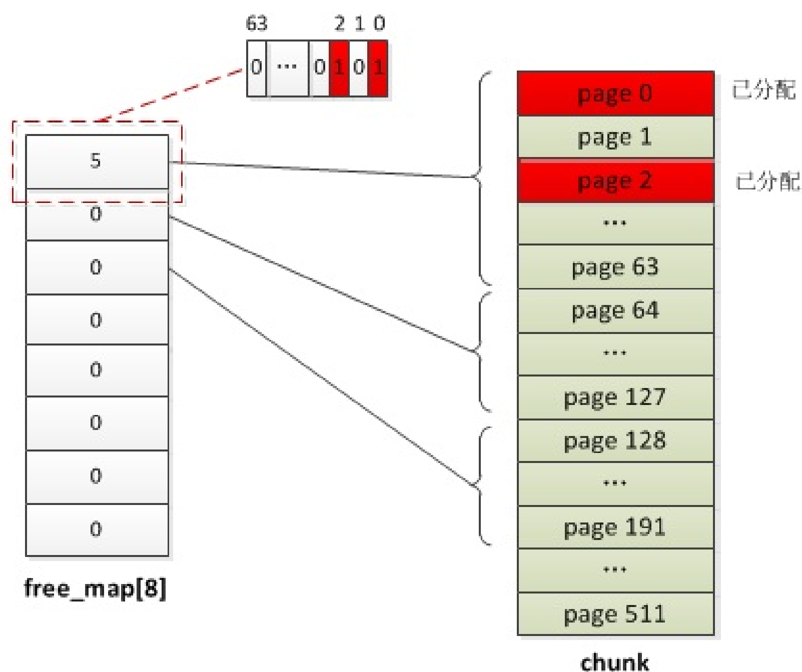
用 `长整形` 的各bit位来记录的（实际就是bitmap），不同位数的机器长整形大小不同，因此在32、64位下16或8个长整形就够512bit了(每个byte等于8bit，长整形为4byte或8byte)，当然这么做并仅仅是节省空间，更重要的作用是可以提高查询效率 。

```
typedef zend_ulong zend_mm_bitset;    /* 4-byte or 8-byte intege
r */
#define ZEND_MM_BITSET_LEN      (sizeof(zend_mm_bitset) * 8)
   /* 32 or 64 */
#define ZEND_MM_PAGE_MAP_LEN    (ZEND_MM_PAGES / ZEND_MM_BITSET_
LEN) /* 16 or 8 */

typedef zend_mm_bitset zend_mm_page_map[ZEND_MM_PAGE_MAP_LEN];
   /* 64B */
```

`heap->free_map` 实际就是：**zend_ulong free_map[16 or 8]**，以 **free_map[8]** 为例，数组中的8个数字分别表示：0-63、64-127、128-191、192-255、256-319、320-383、384-447、448-511 page的分配与否，比如当前chunk的page 0、page 2已经分配，则: `free_map[0] = 5` :

```
//5:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 0
0000101
```



接下来看下 `zend_mm_alloc_pages` 的操作：

```
static void *zend_mm_alloc_pages(zend_mm_heap *heap, int pages_count ZEND_FILE_LINE_DC ZEND_FILE_LINE_ORIG_DC)
{
    zend_mm_chunk *chunk = heap->main_chunk;
    int page_num, len;

    //从第一个chunk开始查找可用page
    while (1) {
        //当前chunk剩余page总数已不够
        if (UNEXPECTED(chunk->free_pages < pages_count)) {
            goto not_found;
        }else{ //查找当前chunk是否有pages_count个连续可用的page
            int best = -1; //已找到可用page起始页
            int best_len = ZEND_MM_PAGES; //已找到chunk的page间隙大小，这个值尽可能接近page_count
            int free_tail = chunk->free_tail;
            zend_mm_bitset *bitset = chunk->free_map;
            zend_mm_bitset tmp = *(bitset++); // zend_mm_bitset tmp = *bitset;  bitset++ 这里是复制出的，不会影响free_map
            int i = 0;

            //下面就是查找最优page的过程,稍后详细分析
            //find best page
```

```
        }

not_found:
        if (chunk->next == heap->main_chunk) { //是否已到最后一个ch
unk
get_chunk:
            ...
        }else{
            chunk = chunk->next;
        }
    }

found: //找到可用page，page编号为page_num至(page_num + pages_count)
    /* mark run as allocated */
    chunk->free_pages -= pages_count;
    zend_mm_bitset_set_range(chunk->free_map, page_num, pages_co
unt); //将page_num至(page_num + pages_count)page的bit标识位设置为已
分配
    chunk->map[page_num] = ZEND_MM_LRUN(pages_count); //map为两个
值的组合值，首先表示当前page属于哪种类型，其次表示包含的page页数
    if (page_num == chunk->free_tail) {
        chunk->free_tail = page_num + pages_count;
    }
    return ZEND_MM_PAGE_ADDR(chunk, page_num);
}
```

查找过程就是从第一个chunk开始搜索，如果当前chunk没有合适的则进入下一chunk，如果直到最后都没有找到则新创建一个chunk。

注意：查找page的过程并不仅仅是够数即可，这里有一个标准是：申请的一个或多个的**page**要尽可能的填满**chunk**的空隙 ，也就是说如果当前chunk有多块内存满足需求则会选择最合适的那块，而合适的标准前面提到的那个。

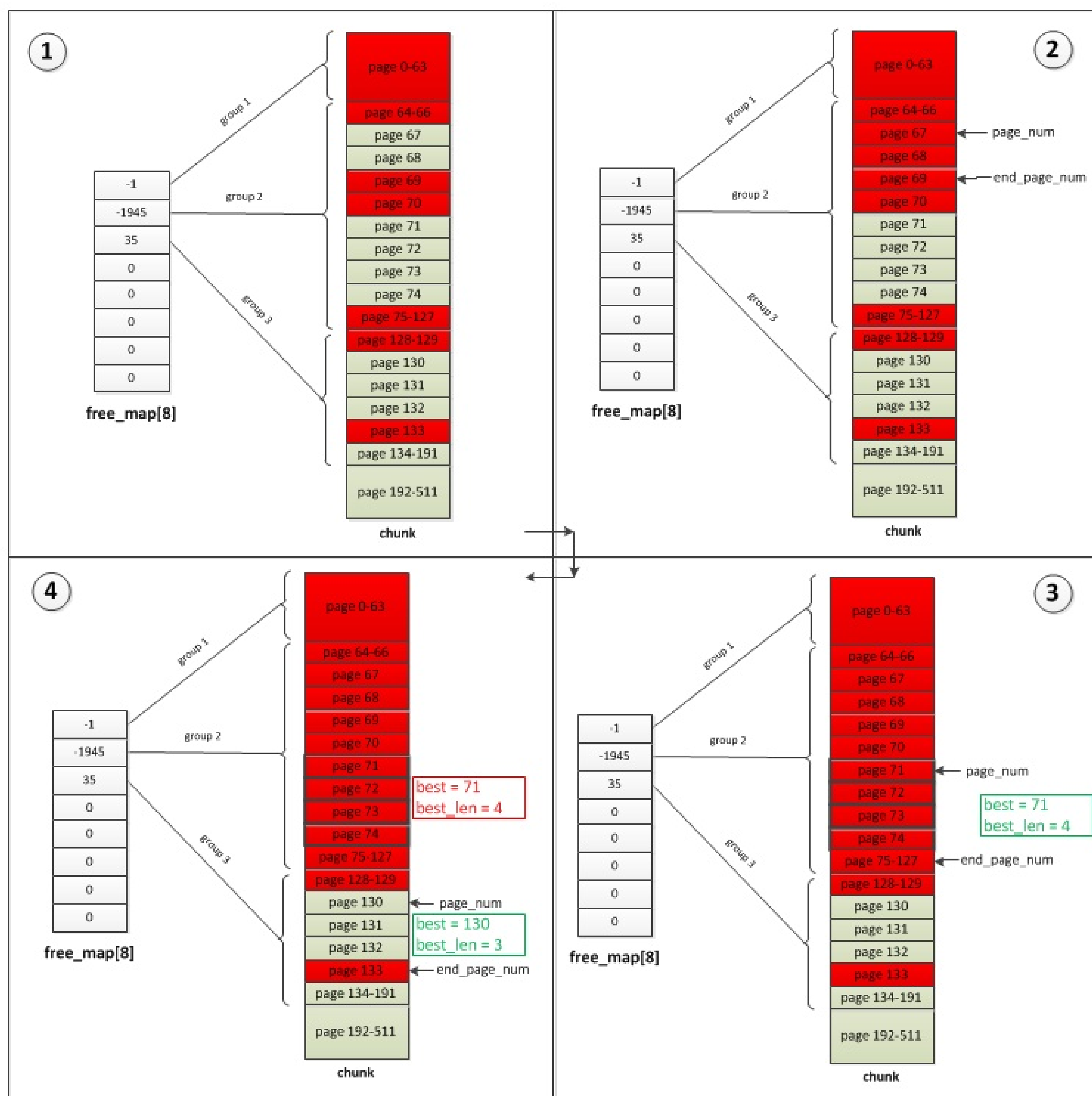
最优**page**的检索过程 ：

- **step1:** 首先从第一个page分组(page 0-63)开始检查，如果当前分组无可用page(即free_map[x] = -1)则进入下一分组，直到当前分组有空闲page，然后进入step2
- **step2:** 当前分组有可用page，首先找到第一个可用page的位置，记作page*num*，接着从***page_num***开始向下找第一个已分配*page*的位置，记作

endpagenum，这个地方需要注意，_如果当前分组剩下的*page*都是可用的则会进入下一分组接着搜索，直到找到为止，这里还会借助chunk->free_tail避免无谓的查找到最后分组

- **step3:** 根据上一步找到的page_num、end_page_num可计算得到当前可用内存块大小为len个page，然后与申请的page页数(page_count)比较
    - **step3.1:** 如果len=page_count则表示找到的内存块符合申请条件且非常完美，直接从page_num开始分配page_count个page
    - **step3.2:** 如果len>page_count则表示找到的内存块符合条件且空间很充裕，暂且记录下len、page_num，然后继续向下搜索，如果有更合适的则用更合适的替代
    - **step3.3:** 如果len<page_count则表示当前内存块不够申请的大小，不符合条件，然后将这块空间的全部page设置为已分配(这样下一轮检索就不会再次找到它了)，接着从step1重新检索

下面从一个例子具体看下，以64bit整形为例，假如当前page分配情况如下图-(1) (group1全部已分配;group2中page 67-68、71-74未分配，其余都已分配;group3中除page 128-129、133已分配外其余都未分配)，现在要申请3个page：



检索过程：

- a.首先会直接跳过group1，直接到group2检索
- b.在group2中找到第一个可用page位置：67，然后向下找第一个不可用page位置：69，找到的可用内存块长度为2，小于3，表示此内存块不可用，这时会将page 67-68标识为已分配，图-(2)
- c.接着再次在group2中查找到第一个可用page位置：71,然后向下找到第一个不可用page位置：75,内存块长度为4，大于3，表示找到一个符合的位置，虽然已经找到可用内存块但并不"完美"，先将这个并不完美的page_num及len保存到best、best_len，如果后面没有比它更完美的就用它了，然后将page 71-74标示为已分配，图-(3)
- d.再次检索，发现group2已无可用page，进入group3，找到可用内存位置：

page 130-132，大小比c中找到的合适，所以最终返回的page就是130-132，图-(4)

page分配完成后会将free_map对应整数的bit位从page_num至(page_num+page_count)置为1，同时将chunk->map[page_num]置为 `ZEND_MM_LRUN(pages_count)` ，表示page_num至(page_num+page_count)这些page是被Large分配占用的。

5.1.3.3 Small分配

small内存指的是小于(3/4 page_size)的内存，这些内存首先也是申请了1个或多个page，然后再将这些page按固定大小切割了，所以第一步与上一节Large分配完全相同。

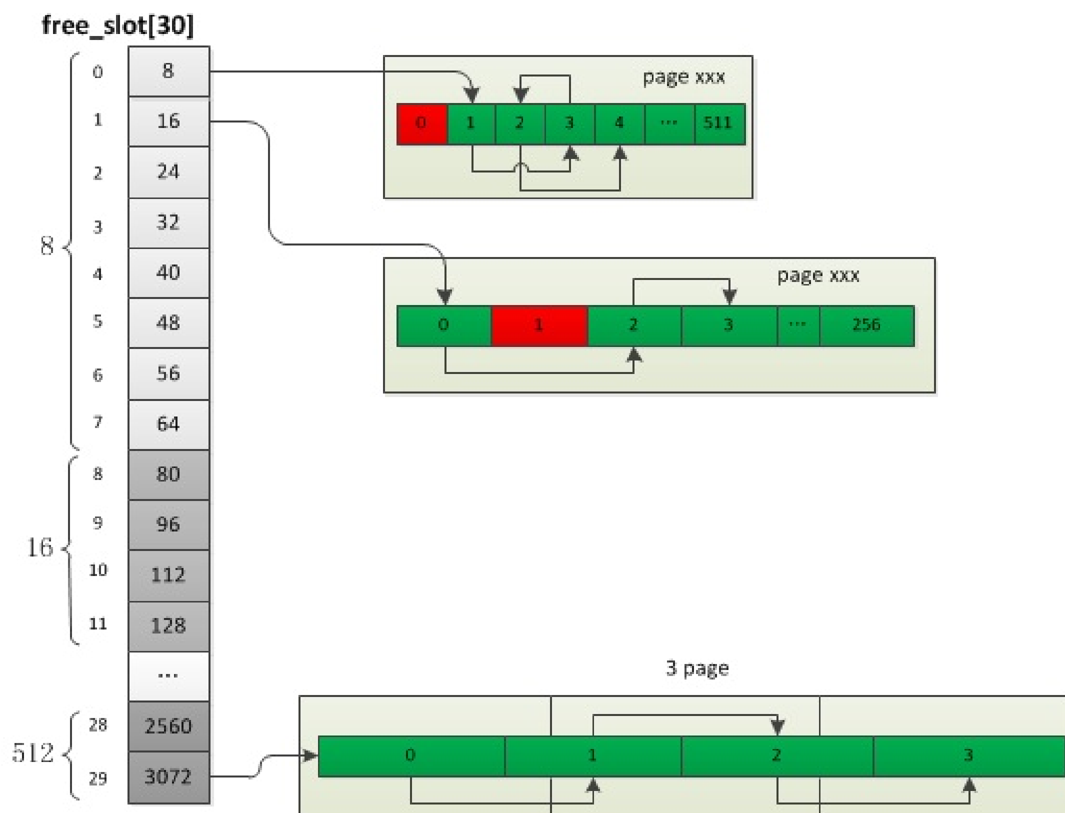
small内存总共有30种固定大小的规格：8,16,24,32,40,48,56,64,80,96,112,128 ... 1792,2048,2560,3072 Byte，我们把这称之为slot，这些slot的大小是有规律的:最小的slot大小为8byte，前8个slot依次递增**8byte**，后面每隔4个递增值乘以2，即 `slot 0-7递增8byte、8-11递增16byte、12-15递增32byte、16-19递增32byte、20-23递增128byte、24-27递增256byte、28-29递增512byte` ，每种大小的slot占用的page数分别是：slot 0-15各占1个page、slot 16-29依次占5, 3, 1, 1, 5, 3, 2, 2, 5, 3, 7, 4, 5, 3个page，这些值定义在 `zend_alloc_sizes.h` 中：

```
/* num, size, count, pages */
#define ZEND_MM_BINS_INFO(_, x, y) \
    _( 0,    8,  512, 1, x, y) \ //四个值的含义依次是：slot编号、slo
t大小、slot数量、占用page数
    _( 1,   16,  256, 1, x, y) \
    _( 2,   24,  170, 1, x, y) \
    _( 3,   32,  128, 1, x, y) \
    _( 4,   40,  102, 1, x, y) \
    _( 5,   48,   85, 1, x, y) \
    _( 6,   56,   73, 1, x, y) \
    _( 7,   64,   64, 1, x, y) \
    _( 8,   80,   51, 1, x, y) \
    _( 9,   96,   42, 1, x, y) \
    _(10,  112,   36, 1, x, y) \
    _(11,  128,   32, 1, x, y) \
    _(12,  160,   25, 1, x, y) \
    _(13,  192,   21, 1, x, y) \
    _(14,  224,   18, 1, x, y) \
    _(15,  256,   16, 1, x, y) \
    _(16,  320,   64, 5, x, y) \
    _(17,  384,   32, 3, x, y) \
    _(18,  448,    9, 1, x, y) \
    _(19,  512,    8, 1, x, y) \
    _(20,  640,   32, 5, x, y) \
    _(21,  768,   16, 3, x, y) \
    _(22,  896,    9, 2, x, y) \
    _(23, 1024,    8, 2, x, y) \
    _(24, 1280,   16, 5, x, y) \
    _(25, 1536,    8, 3, x, y) \
    _(26, 1792,   16, 7, x, y) \
    _(27, 2048,    8, 4, x, y) \
    _(28, 2560,    8, 5, x, y) \
    _(29, 3072,    4, 3, x, y)
```

small内存的分配过程：

- **step1:** 首先根据申请内存的大小在heap->free_slot中找到对应的slot规格bin_num，如果当前slot为空则首先分配对应的page，然后将这些page内存按slot大小切割为zend_mm_free_slot单向链表，free_slot[bin_num]始终指向第

一个可用的slot
- **step2:** 如果申请内存大小对应的的slot链表不为空则直接返回free_slot[bin_num]，然后将free_slot[bin_num]指向下一个空闲位置free_slot[bin_num]->next_free_slot
- **step3:** 释放内存时先将此内存的next_free_slot指向free_slot[bin_num]，然后将free_slot[bin_num]指向释放的内存，也就是将释放的内存插到链表头部



5.1.4 系统内存分配

上面介绍了三种内存分配的过程，内存池实际只是在系统内存上面做了一些工作，尽可能减少系统内存的分配次数，接下来简单看下系统内存的分配。

chunk、page、slot三种内存粒度中chunk的分配是直接向系统申请的，这里调用的并不是malloc（这只是glibc实现的内存操作，并不是操作系统的，zend的内存池实际跟malloc的角色相同），而是mmap：

```
static void *zend_mm_mmap(size_t size)
{
    ...

//hugepage支持
#ifdef MAP_HUGETLB
        if (zend_mm_use_huge_pages && size == ZEND_MM_CHUNK_SIZE
) {
            ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_P
RIVATE | MAP_ANON | MAP_HUGETLB, -1, 0);
            if (ptr != MAP_FAILED) {
                return ptr;
            }
        }
#endif

    ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
 MAP_ANON, -1, 0);

    if (ptr == MAP_FAILED) {
#if ZEND_MM_ERROR
        fprintf(stderr, "\nmmap() failed: [%d] %s\n", errno, str
error(errno));
#endif
        return NULL;
    }
    return ptr;
}
```

HugePage的支持就是在这个地方提现的，详细的可以看下鸟哥的这篇文章：http://www.laruence.com/2015/10/02/3069.html。

『关于Hugepage是啥，简单的说下就是默认的内存是以4KB分页的，而虚拟地址和内存地址是需要转换的，而这个转换是要查表的，CPU为了加速这个查表过程都会内建TLB（Translation Lookaside Buffer），显而易见如果虚拟页越小，表里的条目数也就越多，而TLB大小是有限的，条目数越多TLB的Cache Miss也就会越高，所以如果我们能启用大内存页就能间接降低这个TLB Cache Miss』

5.1.5 内存释放

内存的释放主要是efree操作，与三种分配一一对应，过程也比较简单：

```
#define efree(ptr)                                  _efree((ptr) ZEND_FI
LE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define efree_large(ptr)                            _efree_large((ptr) Z
END_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)
#define efree_huge(ptr)                             _efree_huge((ptr) ZE
ND_FILE_LINE_CC ZEND_FILE_LINE_EMPTY_CC)

ZEND_API void ZEND_FASTCALL _efree(void *ptr ZEND_FILE_LINE_DC Z
END_FILE_LINE_ORIG_DC)
{
    zend_mm_free_heap(AG(mm_heap), ptr ZEND_FILE_LINE_RELAY_CC Z
END_FILE_LINE_ORIG_RELAY_CC);
}

static zend_always_inline void zend_mm_free_heap(zend_mm_heap *h
eap, void *ptr ZEND_FILE_LINE_DC ZEND_FILE_LINE_ORIG_DC)
{
    //根据内存地址及对齐值判断内存地址偏移量是否为0，是的话只有huge情况符合
，page、slot分配出的内存地>址偏移量一定是>=ZEND_MM_CHUNK_SIZE的，因为第
一页始终被chunk自身结构占用，不可能分配出去
    //offset就是ptr距离当前chunk起始位置的偏移量
    size_t page_offset = ZEND_MM_ALIGNED_OFFSET(ptr, ZEND_MM_CHU
NK_SIZE);

    if (UNEXPECTED(page_offset == 0)) {
        if (ptr != NULL) {
            //释放huge内存，从huge_list中删除
            zend_mm_free_huge(heap, ptr ZEND_FILE_LINE_RELAY_CC
ZEND_FILE_LINE_ORIG_RELAY_CC);
        }
    } else { //page或slot，根据chunk->map[]值判断当前page的分配类型
        //根据ptr获取chunk的起始位置
        zend_mm_chunk *chunk = (zend_mm_chunk*)ZEND_MM_ALIGNED_B
ASE(ptr, ZEND_MM_CHUNK_SIZE);
        int page_num = (int)(page_offset / ZEND_MM_PAGE_SIZE);
        zend_mm_page_info info = chunk->map[page_num];
```

```
        ZEND_MM_CHECK(chunk->heap == heap, "zend_mm_heap corrupt
ed");
        if (EXPECTED(info & ZEND_MM_IS_SRUN)) {
            zend_mm_free_small(heap, ptr, ZEND_MM_SRUN_BIN_NUM(i
nfo)); //slot的释放上一节已经介绍过，就是个普通的链表插入操作
        } else /* if (info & ZEND_MM_IS_LRUN) */ {
            int pages_count = ZEND_MM_LRUN_PAGES(info);

            ZEND_MM_CHECK(ZEND_MM_ALIGNED_OFFSET(page_offset, ZE
ND_MM_PAGE_SIZE) == 0, "zend_mm_heap corrupted");
            //释放page，将free_map中的标识位设置为未分配
            zend_mm_free_large(heap, chunk, page_num, pages_coun
t);
        }
    }
}
```

释放的内存地址可能是chunk中间的任意位置，因为chunk分配时是按照ZEND_MM_CHUNK_SIZE对齐的，也就是chunk的起始内存地址一定是ZEND_MM_CHUNK_SIZE的整数倍，所以可以根据chunk上的任意位置知道chunk的起始位置。

释放page的过程有一个地方值得注意，如果释放后发现当前chunk所有page都已经被释放则可能会释放所在chunk，还记得heap->cached_chunks吗？内存池会维持一定的chunk数，每次释放并不会直接销毁而是加入到cached_chunks中，这样下次申请chunk时直接就用了，同时为了防止占用过多内存，cached_chunks会根据每次request请求计算的chunk使用均值保证其维持在一定范围内。

每次request请求结束会对内存池进行一次清理，检查cache的chunk数是否超过均值，超过的话就进行清理，具体的操作：`zend_mm_shutdown`，这里不再展开。

5.2 垃圾回收

5.2.1 垃圾的产生

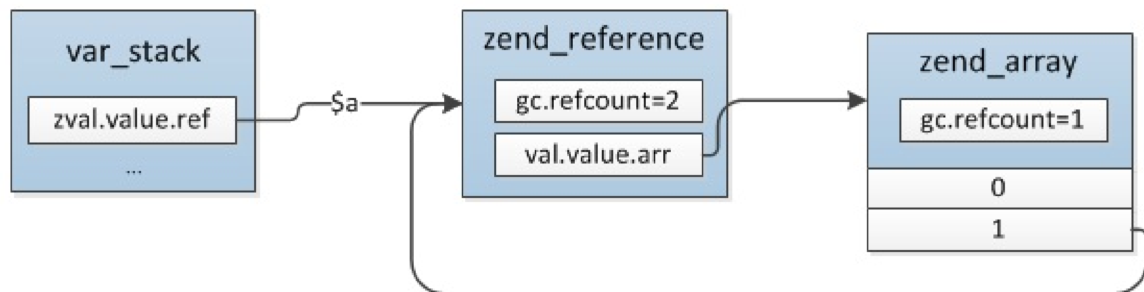
前面已经介绍过PHP变量的内存管理，即引用计数机制，当变量赋值、传递时并不会直接硬拷贝，而是增加value的引用数，unset、return等释放变量时再减掉引用数，减掉后如果发现refcount变为0则直接释放value，这是变量的基本gc过程，PHP正是通过这个机制实现的自动垃圾回收，但是有一种情况是这个机制无法解决的，从而因变量无法回收导致内存始终得不到释放，这种情况就是循环引用，简单的描述就是变量的内部成员引用了变量自身，比如数组中的某个元素指向了数组，这样数组的引用计数中就有一个来自自身成员，试图释放数组时因为其refcount仍然大于0而得不到释放，而实际上已经没有任何外部引用了，这种变量不可能再被使用，所以PHP引入了另外一个机制用来处理变量循环引用的问题。

下面看一个数组循环引用的例子：

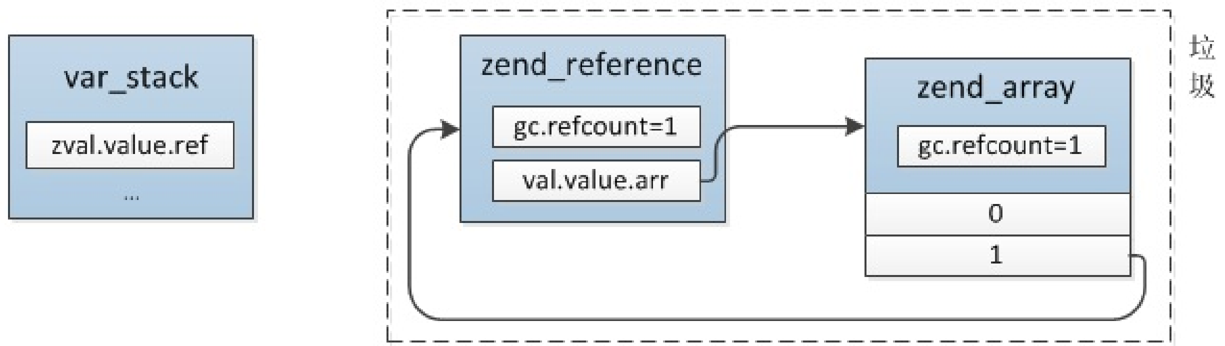
```
$a = [1];
$a[] = &$a;

unset($a);
```

`unset($a)` 之前引用关系：



注意这里$a的类型在 `&` 操作后已经转为引用， `unset($a)` 之后：



可以看到，`unset($a)`之后由于数组中有子元素指向`$a`，所以`refcount = 1`，此时是无法通过正常的gc机制回收的，但是$a已经已经没有任何外部引用了，所以这种变量就是垃圾，垃圾回收器要处理的就是这种情况，这里明确两个准则：

> 1) 如果一个变量value的refcount减少到0， 那么此value可以被释放掉，不属于垃圾
>
> 2) 如果一个变量value的refcount减少之后大于0，那么此zval还不能被释放，此zval可能成为一个垃圾

针对第一个情况GC不会处理，只有第二种情况GC才会将变量收集起来。另外变量是否加入垃圾检查buffer并不是根据zval的类型判断的，而是与前面介绍的是否用到引用计数一样通过`zval.u1.type_flag`记录的，只有包含`IS_TYPE_COLLECTABLE`的变量才会被GC收集。

目前垃圾只会出现在array、object两种类型中，数组的情况上面已经介绍了，object的情况则是成员属性引用对象本身导致的，其它类型不会出现这种变量中的成员引用变量自身的情况，所以垃圾回收只会处理这两种类型的变量。

```
#define IS_TYPE_COLLECTABLE
```

```
|     type       | collectable |
+----------------+-------------+
|simple types    |             |
|string          |             |
|interned string |             |
|array           |      Y      |
|immutable array |             |
|object          |      Y      |
|resource        |             |
|reference       |             |
```

5.2.2 回收过程

如果当变量的refcount减少后大于0，PHP并不会立即进行对这个变量进行垃圾鉴定，而是放入一个缓冲buffer中，等这个buffer满了以后(10000个值)再统一进行处理，加入buffer的是变量zend_value的 `zend_refcounted_h` :

```
typedef struct _zend_refcounted_h {
    uint32_t         refcount; //记录zend_value的引用数
    union {
        struct {
            zend_uchar    type,  //zend_value的类型,与zval.u1.typ
e一致
            zend_uchar    flags,
            uint16_t      gc_info //GC信息，垃圾回收的过程会用到
        } v;
        uint32_t type_info;
    } u;
} zend_refcounted_h;
```

一个变量只能加入一次buffer，为了防止重复加入，变量加入后会把 `zend_refcounted_h.gc_info` 置为 `GC_PURPLE` ，即标为紫色，下次refcount减少时如果发现已经加入过了则不再重复插入。垃圾缓存区是一个双向链表，等到缓存区满了以后则启动垃圾检查过程：遍历缓存区，再对当前变量的所有成员进行遍历，然后把成员的refcount减1(如果成员还包含子成员则也进行递归遍历，其实就是深度优先的遍历)，最后再检查当前变量的引用，如果减为了0则为垃圾。这个

算法的原理很简单，垃圾是由于成员引用自身导致的，那么就对所有的成员减一遍引用，结果如果发现变量本身refcount变为了0则就表明其引用全部来自自身成员。具体的过程如下：

(1) 从buffer链表的roots开始遍历，把当前value标为灰色(zend_refcounted_h.gc_info置为GC_GREY)，然后对当前value的成员进行深度优先遍历，把成员value的refcount减1，并且也标为灰色；

(2) 重复遍历buffer链表，检查当前value引用是否为0，为0则表示确实是垃圾，把它标为白色(GC_WHITE)，如果不为0则排除了引用全部来自自身成员的可能，表示还有外部的引用，并不是垃圾，这时候因为步骤(1)对成员进行了refcount减1操作，需要再还原回去，对所有成员进行深度遍历，把成员refcount加1，同时标为黑色；

(3) 再次遍历buffer链表，将非GC_WHITE的节点从roots链表中删除，最终roots链表中全部为真正的垃圾，最后将这些垃圾清除。

5.2.3 垃圾收集的内部实现

接下来我们简单看下垃圾回收的内部实现，垃圾收集器的全局数据结构：

```
typedef struct _zend_gc_globals {
    zend_bool         gc_enabled; //是否启用gc
    zend_bool         gc_active;  //是否在垃圾检查过程中
    zend_bool         gc_full;    //缓存区是否已满

    gc_root_buffer   *buf;   //启动时分配的用于保存可能垃圾的缓存区
    gc_root_buffer    roots; //指向buf中最新加入的一个可能垃圾
    gc_root_buffer   *unused;//指向buf中没有使用的buffer
    gc_root_buffer   *first_unused; //指向buf中第一个没有使用的buffer

    gc_root_buffer   *last_unused; //指向buf尾部

    gc_root_buffer    to_free;  //待释放的垃圾
    gc_root_buffer   *next_to_free;

    uint32_t gc_runs;   //统计gc运行次数
    uint32_t collected; //统计已回收的垃圾数
} zend_gc_globals;

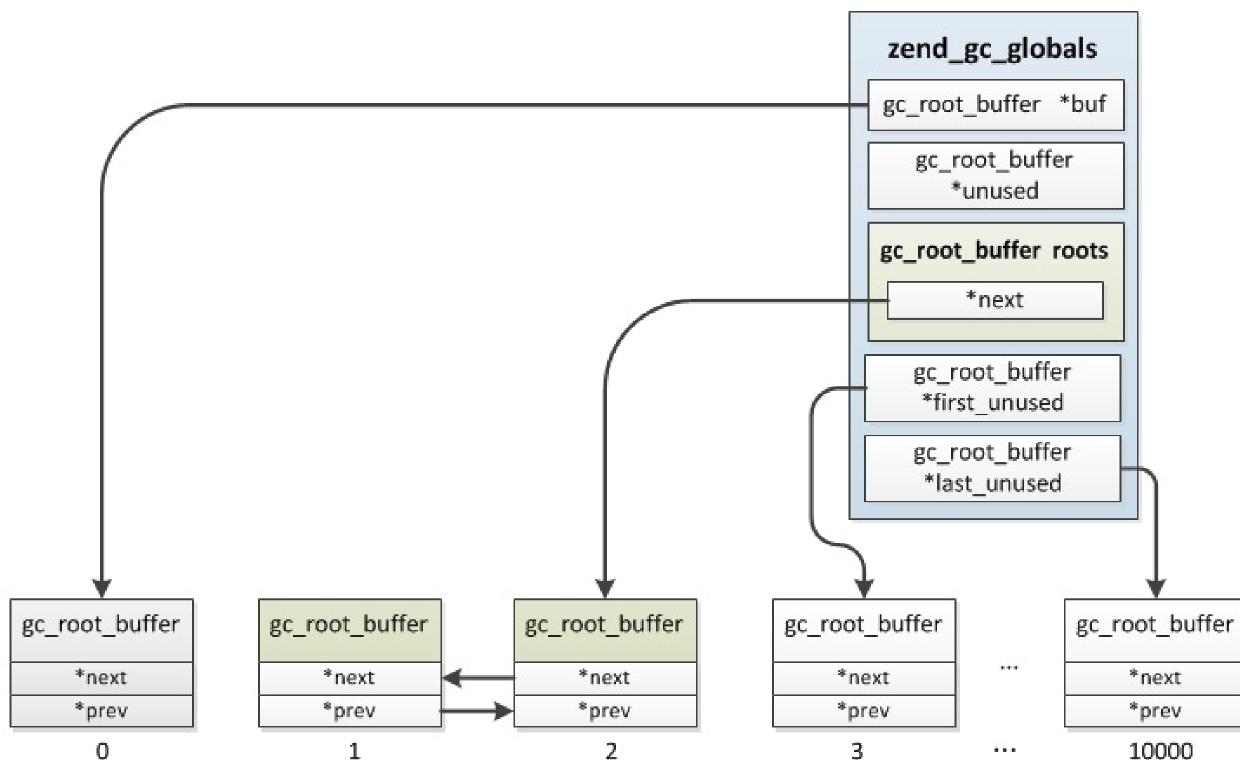
typedef struct _gc_root_buffer {
    zend_refcounted          *ref; //每个zend_value的gc信息
    struct _gc_root_buffer   *next;
    struct _gc_root_buffer   *prev;
    uint32_t                 refcount;
} gc_root_buffer;
```

`zend_gc_globals` 是垃圾回收过程中主要用到的一个结构，用来保存垃圾回收器的所有信息，比如垃圾缓存区； `gc_root_buffer` 用来保存每个可能是垃圾的变量，它实际就是整个垃圾收集buffer链表的元素，当GC收集一个变量时会创建一个 `gc_root_buffer` ，插入链表。

`zend_gc_globals` 这个结构中有几个关键成员：

- **(1)buf:** 前面已经说过，当refcount减少后如果大于0那么就会将这个变量的value加入GC的垃圾缓存区，buf就是这个缓存区，它实际是一块连续的内存，在GC初始化时一次性分配了10001个gc_root_buffer，插入变量时直接从buf中取出可用节点；
- **(2)roots:** 垃圾缓存链表的头部，启动GC检查的过程就是从roots开始遍历的；

- **(3)first_unused:** 指向buf中第一个可用的节点，初始化时这个值为1而不是0，因为第一个gc_root_buffer保留没有使用，有元素插入roots时如果first_unused还没有到达buf的尾部则返回first_unused给最新的元素，然后first_unused++，直到last_unused，比如现在已经加入了2个可能的垃圾变量，则对应的结构：



- **(4)last_unused:** 与first_unused类似，指向buf末尾
- **(5)unused:** GC收集变量时会依次从buf中获取可用的gc_root_buffer，这种情况直接取first_unused即可，但是有些变量加入垃圾缓存区之后其refcount又减为0了，这种情况就需要从roots中删掉，因为它不可能是垃圾，这样就导致roots链表并不是像buf分配的那样是连续的，中间会出现一些开始加入后面又删除的节点，这些节点就通过unused串成一个单链表，unused指向链表尾部，下次有新的变量插入roots时优先使用unused的这些节点，其次才是first_unused的，举个例子：```php //示例1： $a = array(); //$a -> zend_array(refcount=1) $b = $a; //$a -> zend_array(refcount=2)

```
        //$b ->
```

unset($b); //此时zend_array(refcount=1)，因为refoucnt>0所以加入gc的垃圾缓存区：roots unset($a); //此时zend_array(refcount=0)且gc_info为GC_PURPLE，则从roots链表中删掉

````
假如`unset($b)`时插入的是buf中第1个位置，那么`unset($a)`后对应的结构：

![](../img/zend_gc_2.png)

如果后面再有变量加入GC垃圾缓存区将优先使用第1个。

此GC机制可以通过php.ini中`zend.enable_gc`设置是否开启，如果开启则在php.ini解析后调用`gc_init()`进行GC初始化：
```c
ZEND_API void gc_init(void)
{
    if (GC_G(buf) == NULL && GC_G(gc_enabled)) {
        //分配buf缓存区内存，大小为GC_ROOT_BUFFER_MAX_ENTRIES(10001)，其中第1个保留不被使用
        GC_G(buf) = (gc_root_buffer*) malloc(sizeof(gc_root_buffer) * GC_ROOT_BUFFER_MAX_ENTRIES);
        GC_G(last_unused) = &GC_G(buf)[GC_ROOT_BUFFER_MAX_ENTRIES];
        //进行GC_G的初始化，其中：GC_G(first_unused) = GC_G(buf) + 1;从第2个开始的，第1个保留
        gc_reset();
    }
}
````

在PHP的执行过程中，如果发现array、object减掉refcount后大于0则会调用 `gc_possible_root()` 将zend_value的gc头部加入GC垃圾缓存区：

```c
ZEND_API void ZEND_FASTCALL gc_possible_root(zend_refcounted *ref)
{
    gc_root_buffer *newRoot;

    //插入的节点必须是GC_BLACK，防止重复插入
    ZEND_ASSERT(EXPECTED(GC_REF_GET_COLOR(ref) == GC_BLACK));

    newRoot = GC_G(unused); //先看下unused中有没有可用的
    if (newRoot) {
        //有的话先用unused的，然后将GC_G(unused)指向单链表的下一个
        GC_G(unused) = newRoot->prev;
```
```

```
    } else if (GC_G(first_unused) != GC_G(last_unused)) {
        //unused没有可用的，且buf中还有可用的
        newRoot = GC_G(first_unused);
        GC_G(first_unused)++;
    } else {
        //buf缓存区已满，这时需要启动垃圾检查程序了，遍历roots，将真正的
垃圾释放
        //垃圾回收的动作就是在这触发的
        if (!GC_G(gc_enabled)) {
            return;
        }
        ...

        //启动垃圾回收过程
        gc_collect_cycles(); //即：zend_gc_collect_cycles()
        ...
    }

    //将插入的ref标为紫色，防止重复插入
    GC_TRACE_SET_COLOR(ref, GC_PURPLE);
    //注意：gc_info不仅仅只有颜色的信息，还会记录当前gc_root_buffer在整
个buf中的位置
    //这样做的目的是可以直接根据zend_value的gc信息取到它的gc_root_buffe
r，便于进行删除操作
    GC_INFO(ref) = (newRoot - GC_G(buf)) | GC_PURPLE;
    newRoot->ref = ref;

    //GC_G(roots).next指向新插入的元素
    newRoot->next = GC_G(roots).next;
    newRoot->prev = &GC_G(roots);
    GC_G(roots).next->prev = newRoot;
    GC_G(roots).next = newRoot;
}
```

同一个zend_value只会插入一次，再次插入时如果发现其gc_info不是GC_BLACK则直接跳过。另外像上面示例1的情况，插入后如果后面发现其refcount减为0了则表明它可以直接被回收掉，这时需要把这个节点从roots链表中删除，删除的操作通过 `GC_REMOVE_FROM_BUFFER()` 宏操作：

```
#define GC_REMOVE_FROM_BUFFER(p) do { \
    zend_refcounted *_p = (zend_refcounted*)(p); \
        if (GC_ADDRESS(GC_INFO(_p))) { \
            gc_remove_from_buffer(_p); \
        } \
} while (0)

ZEND_API void ZEND_FASTCALL gc_remove_from_buffer(zend_refcounte
d *ref)
{
    gc_root_buffer *root;

    //GC_ADDRESS就是获取节点在缓存区中的位置，因为删除时输入是zend_refco
unted
    //而缓存链表的节点类型是gc_root_buffer
    root = GC_G(buf) + GC_ADDRESS(GC_INFO(ref));
    if (GC_REF_GET_COLOR(ref) != GC_BLACK) {
        GC_TRACE_SET_COLOR(ref, GC_PURPLE);
    }
    GC_INFO(ref) = 0;
    GC_REMOVE_FROM_ROOTS(root); //双向链表的删除操作
    ...
}
```

插入时如果发现垃圾缓存链表已经满了，则会启动垃圾回收过程： `zend_gc_collect_cycles()` ，这个过程会对之前插入缓存区的变量进行判断是否是循环引用导致的真正的垃圾，如果是垃圾则会进行回收，回收的过程前面已经介绍过:

```
ZEND_API int zend_gc_collect_cycles(void)
{
    ...
    //(1)遍历roots链表，对当前节点value的所有成员(如数组元素、成员属性)进行深度优先遍历把成员refcount减1
    gc_mark_roots();

    //(2)再次遍历roots链表，检查各节点当前refcount是否为0，是的话标为白色，表示是垃圾，不是的话需要对还原(1)，把refcount再加回去
    gc_scan_roots();

    //(3)将roots链表中的非白色节点删除，之后roots链表中全部是真正的垃圾，将垃圾链表转到to_free等待释放
    count = gc_collect_roots(&gc_flags, &additional_buffer);
    ...

    //(4)释放垃圾
    current = to_free.next;
    while (current != &to_free) {
        p = current->ref;
        GC_G(next_to_free) = current->next;
        if ((GC_TYPE(p) & GC_TYPE_MASK) == IS_OBJECT) {
            //调用free_obj释放对象
            obj->handlers->free_obj(obj);
            ...
        } else if ((GC_TYPE(p) & GC_TYPE_MASK) == IS_ARRAY) {
            //释放数组
            zend_array *arr = (zend_array*)p;

            GC_TYPE(arr) = IS_NULL;
            zend_hash_destroy(arr);
        }
        current = GC_G(next_to_free);
    }
    ...
}
```

各步骤具体的操作不再详细展开，这里单独说明下value成员的遍历，array比较好理解，所有成员都在arData数组中，直接遍历arData即可，如果各元素仍是array、object或者引用则一直递归进行深度优先遍历；object的成员指的成员属性（不包括静态属性、常量，它们属于类而不属于对象），前面介绍对象的实现时曾说过，成员属性除了明确的在类中定义的那些外还可以动态创建，动态属性保存于zend_obejct->properties哈希表中，普通属性保存于zend_object.properties_table数组中，这样以来object的成员就分散在两个位置，那么遍历时是分别遍历吗？答案是否定的。

实际前面已经简单提过，在创建动态属性时会把全部普通属性也加到zend_obejct->properties哈希表中，指向原zend_object.properties_table中的属性，这样一来GC遍历object的成员时就可以像array那样遍历zend_obejct->properties即可，GC获取object成员的操作由get_gc(即：zend_std_get_gc())完成：

```
ZEND_API HashTable *zend_std_get_gc(zval *object, zval **table,
int *n)
{
    if (Z_OBJ_HANDLER_P(object, get_properties) != zend_std_get_
properties) {
        *table = NULL;
        *n = 0;
        return Z_OBJ_HANDLER_P(object, get_properties)(object);
    } else {
        zend_object *zobj = Z_OBJ_P(object);

        if (zobj->properties) {
            //有动态属性
            *table = NULL;
            *n = 0;
            return zobj->properties;
        } else {
            //没有定义过动态属性，返回数组
            *table = zobj->properties_table;
            *n = zobj->ce->default_properties_count;
            return NULL;
        }
    }
}
```

6.1 介绍

在C语言中声明在任何函数之外的变量为全局变量，全局变量为各线程共享，不同的线程引用同一地址空间，如果一个线程修改了全局变量就会影响所有的线程。所以线程安全是指多线程环境下如何安全的获取公共资源。

PHP的SAPI多数是单线程环境，比如cli、fpm、cgi，每个进程只启动一个主线程，这种模式下是不存在线程安全问题的，但是也有多线程的环境，比如Apache，或用户自己嵌入PHP实现的环境，这种情况下就需要考虑线程安全的问题了，因为PHP中有很多全局变量，比如最常见的：EG、CG，如果多个线程共享同一个变量将会冲突，所以PHP为多线程的应用模型提供了一个安全机制：Zend线程安全(Zend Thread Safe, ZTS)。

6.2 线程安全资源管理器

PHP中专门为解决线程安全的问题抽象出了一个线程安全资源管理器(Thread Safe Resource Mananger, TSRM)，实现原理比较简单：既然共用资源这么困难那么就干脆不共用，各线程不再共享同一份全局变量，而是各复制一份，使用数据时各线程各取自己的副本，互不干扰。

6.2.1 基本实现

TSRM核心思想就是为不同的线程分配独立的内存空间，如果一个资源会被多线程使用，那么首先需要预先向TSRM注册资源，然后TSRM为这个资源分配一个唯一的编号，并把这种资源的大小、初始化函数等保存到一个 `tsrm_resource_type` 结构中，各线程只能通过TSRM分配的那个编号访问这个资源；然后当线程拿着这个编号获取资源时TSRM如果发现是第一次请求，则会根据注册时的资源大小分配一块内存，然后调用初始化函数进行初始化，并把这块资源保存下来供这个线程后续使用。

TSRM中通过两个结构分别保存资源信息以及具体的资源：tsrm_resource_type、tsrm_tls_entry，前者是用来记录资源大小、初始化函数等信息的，具体分配资源内存时会用到，而后者用来保存各线程所拥有的全部资源：

```
struct _tsrm_tls_entry {
    void **storage; //资源数组
    int count; //拥有的资源数:storage数组大小
    THREAD_T thread_id; //所属线程id
    tsrm_tls_entry *next;
};

typedef struct {
    size_t size; //资源的大小
    ts_allocate_ctor ctor; //初始化函数
    ts_allocate_dtor dtor;
    int done;
} tsrm_resource_type;
```

每个线程拥有一个`tsrm_tls_entry`结构，当前线程的所有资源保存在storage数组中，下标就是各资源的id。

另外所有线程的`tsrm_tls_entry`结构通过一个数组保存：tsrm_tls_table，这是个全局变量，所以操作这个变量时需要加锁。这个值在TSRM初始化时按照预设置的线程数分配，每个线程的tsrm_tls_entry结构在这个数组中的位置是根据线程id与预设置的线程数(tsrm_tls_table_size)取模得到的，也就是说有可能多个线程保存在tsrm_tls_table同一位置，所以tsrm_tls_entry是个链表，查找资源时首先根据:`线程id % tsrm_tls_table_size`得到一个tsrm_tls_entry，然后开始遍历链表比较thread_id确定是否是当前线程的。

6.2.1.1 初始化

在使用TSRM之前需要主动开启，一般这个步骤在sapi启动时执行，主要工作就是分配tsrm_tls_table、resource_types_table内存以及创建线程互斥锁，下面具体看下TSRM初始化的过程(以pthread为例)：

```
TSRM_API int tsrm_startup(int expected_threads, int expected_res
ources, int debug_level, char *debug_filename)
{
    pthread_key_create( &tls_key, 0 );

    //分配tsrm_tls_table
    tsrm_tls_table_size = expected_threads;
    tsrm_tls_table = (tsrm_tls_entry **) calloc(tsrm_tls_table_s
ize, sizeof(tsrm_tls_entry *));
    ...
    //初始化资源的递增id，注册资源时就是用的这个值
    id_count=0;

    //分配资源类型数组：resource_types_table
    resource_types_table_size = expected_resources;
    resource_types_table = (tsrm_resource_type *) calloc(resourc
e_types_table_size, sizeof(tsrm_resource_type));
    ...
    //创建锁
    tsmm_mutex = tsrm_mutex_alloc();
}
```

6.2.1.2 资源注册

初始化完成各模块就可以各自进行资源注册了，注册后TSRM会给注册的资源分配唯一id，之后对此资源的操作只能依据此id，接下来我们以EG为例具体看下其注册过程。

```
#ifdef ZTS
ZEND_API int executor_globals_id;
#endif

int zend_startup(zend_utility_functions *utility_functions, char
 **extensions)
{
    ...
#ifdef ZTS
    ts_allocate_id(&executor_globals_id, sizeof(zend_executor_gl
obals), (ts_allocate_ctor) executor_globals_ctor, (ts_allocate_d
tor) executor_globals_dtor);

    executor_globals = ts_resource(executor_globals_id);
    ...
#endif
}
```

资源注册调用 `ts_allocate_id()` 完成，此函数有4个参数有，第一个就是定义的资源id指针，注册之后会把分配的id写到这里，第二个是资源类型的大小，EG资源的结构是 `zend_executor_globals` ，所以这个值就是sizeof(zend_executor_globals)，后面两个分别是资源的初始化函数以及销毁函数，因为TSRM并不关心资源的具体类型，分配资源时它只按照size大小分配内存，然后回调各资源自己定义的ctor进行初始化。

```
TSRM_API ts_rsrc_id ts_allocate_id(ts_rsrc_id *rsrc_id, size_t s
ize, ts_allocate_ctor ctor, ts_allocate_dtor dtor)
{
    //加锁，保证各线程串行调用此函数
    tsrm_mutex_lock(tsmm_mutex);

    //分配id，即id_count当前值，然后把id_count加1
    *rsrc_id = TSRM_SHUFFLE_RSRC_ID(id_count++);

    //检查resource_types_table数组当前大小是否已满
    if (resource_types_table_size < id_count) {
        //需要对resource_types_table扩容
        resource_types_table = (tsrm_resource_type *) realloc(re
source_types_table, sizeof(tsrm_resource_type)*id_count);
        ...
        //把数组大小修改新的大小
        resource_types_table_size = id_count;
    }

    //将新注册的资源插入resource_types_table数组，下标就是分配的资源id
    resource_types_table[TSRM_UNSHUFFLE_RSRC_ID(*rsrc_id)].size
= size;
    resource_types_table[TSRM_UNSHUFFLE_RSRC_ID(*rsrc_id)].ctor
= ctor;
    resource_types_table[TSRM_UNSHUFFLE_RSRC_ID(*rsrc_id)].dtor
= dtor;
    resource_types_table[TSRM_UNSHUFFLE_RSRC_ID(*rsrc_id)].done
= 0;
    ...
}
```

到这里并没有结束，所有的资源并不是统一时机注册的，所以注册一个新资源时可能有线程已经分配先前注册的资源了，因此需要对各线程的storage数组进行扩容，否则storage将没有空间容纳新的资源。扩容的过程比较简单：遍历各线程的tsrm_tls_entry，检查storage当时是否有空闲空间，有的话跳过，没有的话则扩展。

```
for (i=0; i<tsrm_tls_table_size; i++) {
    tsrm_tls_entry *p = tsrm_tls_table[i];

    //tsrm_tls_table[i]可能保存着多个线程，需要遍历链表
    while (p) {
        if (p->count < id_count) {
            int j;

            //将storage扩容
            p->storage = (void *) realloc(p->storage, sizeof(void
 *)*id_count);
            //分配并初始化新注册的资源，实际这里只会执行一次，不清楚为什么
用循环
            //另外这里不分配内存也可以，可以放到使用时再去分配
            for (j=p->count; j<id_count; j++) {
                p->storage[j] = (void *) malloc(resource_types_t
able[j].size);
                if (resource_types_table[j].ctor) {
                    //回调初始化函数进行初始化
                    resource_types_table[j].ctor(p->storage[j]);
                }
            }
            p->count = id_count;
        }
        p = p->next;
    }
}
```

最后将锁释放，完成注册。

6.2.1.3 获取资源

资源的id在注册后需要保存下来，根据id可以通过 `ts_resource()` 获取到对应资源的值，比如EG，这里暂不考虑EG宏展开的结果，只分析最底层的根据资源id获取资源的操作。

```
zend_executor_globals *executor_globals;

executor_globals = ts_resource(executor_globals_id);
```

这样获取的 `executor_globals` 值就是各线程分离的了，对它的操作将不会再影响其它线程。根据资源id获取当前线程资源的过程：首先是根据线程id哈希得到当前线程的tsrm_tls_entry在tsrm_tls_table哪个槽中，然后开始遍历比较id，直到找到当前线程的tsrm_tls_entry，这个查找过程是需要加锁的，最后根据资源id从storage中对应位置取出资源的地址，这个时候如果发现当前线程还没有创建此资源则会从resource_types_table根据资源id取出资源注册时的大小、初始化函数，然后分配内存、调用初始化函数进行初始化并插入所属线程的storage中。

```
TSRM_API void *ts_resource_ex(ts_rsrc_id id, THREAD_T *th_id)
{
    THREAD_T thread_id;
    int hash_value;
    tsrm_tls_entry *thread_resources;

    //step 1：获取线程id
    if (!th_id) {
        //获取当前线程通过specific data保存的tsrm_tls_entry，暂时忽略
        thread_resources = tsrm_tls_get();
        if(thread_resources){
            //找到线程的tsrm_tls_entry了
            TSRM_SAFE_RETURN_RSRC(thread_resources->storage, id,
 thread_resources->count); //直接返回
        }
        //pthread_self()，当前线程id
        thread_id = tsrm_thread_id();
    }else{
        thread_id = *th_id;
    }

    //step 2:查找线程tsrm_tls_entry
    tsrm_mutex_lock(tsmm_mutex); //加锁

    //实际就是thread_id % tsrm_tls_table_size
    hash_value = THREAD_HASH_OF(thread_id, tsrm_tls_table_size);
```

```
    //链表头部
    thread_resources = tsrm_tls_table[hash_value];
    if (!thread_resources) {
        //当前线程第一次使用资源还未分配：先分配tsrm_tls_entry
        allocate_new_resource(&tsrm_tls_table[hash_value], threa
d_id);
        //分配完再次调用，这时候将走到下面的分支
        return ts_resource_ex(id, &thread_id);
    }else{
        //遍历查找当前线程的tsrm_tls_entry
        do {
            //找到了
            if (thread_resources->thread_id == thread_id) {
                break;
            }
            if (thread_resources->next) {
                thread_resources = thread_resources->next;
            } else {
                //遍历到最后也没找到，与上面的一致，先分配再查找
                allocate_new_resource(&thread_resources->next, t
hread_id);
                return ts_resource_ex(id, &thread_id);
            }
        } while (thread_resources);
    }
    //解锁
    tsrm_mutex_unlock(tsmm_mutex);

    //step 3：返回资源
    TSRM_SAFE_RETURN_RSRC(thread_resources->storage, id, thread_
resources->count);
}
```

首先是获取线程id，如果没有传的话就是当前线程，然后在tsrm_tls_table中查找当前线程的tsrm_tls_entry，不存在则表示当前线程第一次使用资源，则需要调用 `allocate_new_resource()` 为当前线程分配tsrm_tls_entry，并插入tsrm_tls_table，这个过程还会为当前已注册的所有资源分配内存：

```
static void allocate_new_resource(tsrm_tls_entry **thread_resour
ces_ptr, THREAD_T thread_id)
{
    (*thread_resources_ptr) = (tsrm_tls_entry *) malloc(sizeof(t
srm_tls_entry));
    (*thread_resources_ptr)->storage = NULL;
    //根据已注册资源数分配storage数组大小，注意这里并不是分配为各资源分配
空间
    if (id_count > 0) {
        (*thread_resources_ptr)->storage = (void **) malloc(size
of(void *)*id_count);
    }
    (*thread_resources_ptr)->count = id_count;
    (*thread_resources_ptr)->thread_id = thread_id;

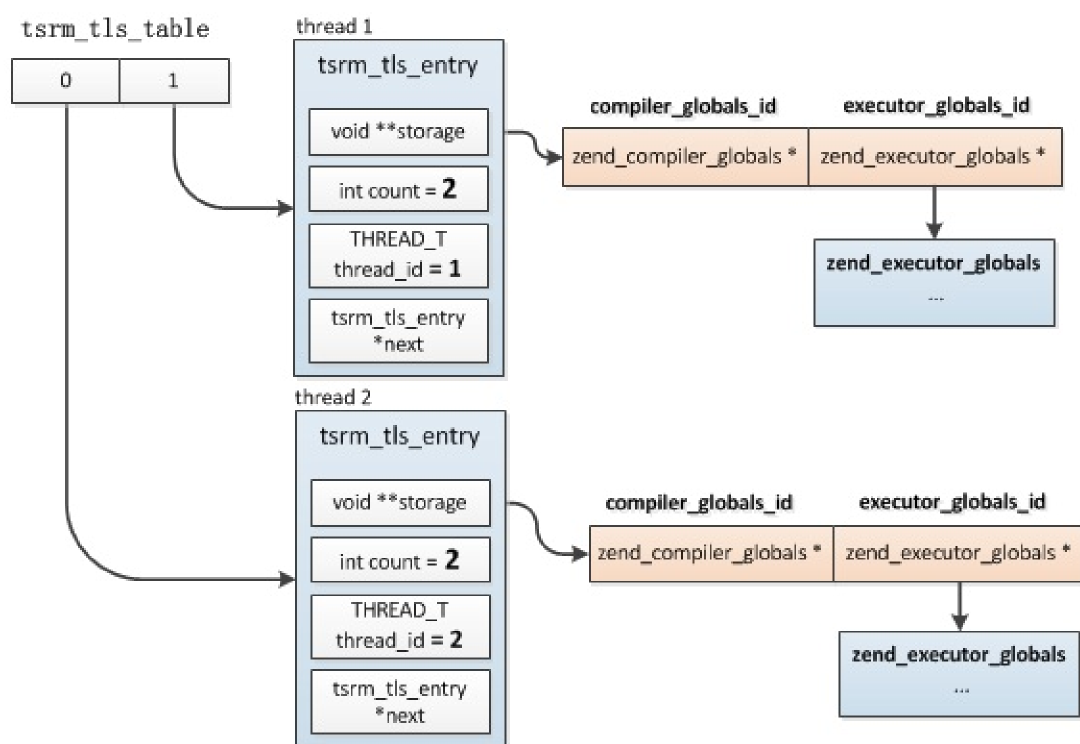
    //将当前线程的tsrm_tls_entry保存到线程本地存储(Thread Local Stora
ge, TLS)
    tsrm_tls_set(*thread_resources_ptr);

    //为全部资源分配空间
    for (i=0; i<id_count; i++) {
        ...
        (*thread_resources_ptr)->storage[i] = (void *) malloc(re
source_types_table[i].size);
        ...
    }
    ...
}
```

这里还用到了一个多线程中经常用到的一个东西：线程本地存储(Thread Local Storage, TLS)，在创建完当前线程的tsrm_tls_entry后会把这个值保存到当前线程的TLS中(即：tsrm_tls_set(*thread_resources_ptr)操作)，这样在 `ts_resource()` 中就可以通过 `tsrm_tls_get()` 直接取到了，节省加锁检索的时间。

> 线程本地存储(**Thread Local Storage, TLS**): 我们知道在一个进程中，所有线程是共享同一个地址空间的。所以，如果一个变量是全局的或者是静态的，那么所有线程访问的是同一份，如果某一个线程对其进行了修改，也就会影响到其他所有的线程。不过我们可能并不希望这样，所以更多的推荐用基于堆栈的自动变量或函数参数来访问数据，因为基于堆栈的变量总是和特定的线程相联系的。TLS在各平台下实现方式不同，主要分为两类：静态TLS、动态TLS，pthread中pthread_setspecific()、pthread_getspecific()的实现就可以认为是动态TLS的实现。

比如tsrm_tls_table_size初始化时设置为了2，当前有2个thread：thread 1、thread 2，假如注册了CG、EG两个资源，则存储结构如下图：



6.2.2 Native-TLS

上一节我们介绍了资源的注册以及根据资源id获取资源的方法，那么PHP内核每次使用对应的资源时难道都需要调用 `ts_resource()` 吗？如果是这样的话那么多次在使用EG时实际都会调一次这个方法，相当于我们需要调用一个函数来获取一个变量，这在性能上是不可接受的，那么有什么办法解决呢？

`ts_resource()` 最核心的操作就是根据线程id获取各线程对应的storage数组，这也是最耗时的部分，至于接下来根据资源id从storage数组读取资源就是普通的内存读取了，这并不影响性能，所以解决上面那个问题的关键就在于 尽可能的减少线程

storage的检索 。这一节我们来分析下PHP是如果解决这个问题的，在介绍PHP7实现方式之前我们先看下PHP5.x的处理方式。

PHP5的解决方式非常简单，我们还是以EG为例，EG在内核中随处可见，不是要减少对各线程storage的检索次数吗，那么我就只要检索过一次就把已获取的storage指针传给接下来调用的函数用，其它函数再一级级往下传，这样一来各函数如果发现storage通过参数传进来了就直接用，无需再检索了，也就是通过层层传递的方式减少解决这个问题的。这样以来岂不是每个函数都得带这么一个参数？调用别的函数也得把这个值带上？是的。即使这个函数自己不用它也得需要这个值，因为有可能调用别的函数的时候其它函数会用。

如果你对PHP5有所了解的话一定经常看到这两个宏：TSRMLS_DC、TSRMLS_CC，这两个宏就是用来传递storage指针的，TSRMLS_DC用在定义函数的参数中，实际上它就是一个普通的参数定义，TSRMLS_CC用在调用函数时，它就是一个普通的变量值，我们看下它的展开结果：

```
#define TSRMLS_DC   , void ***tsrm_ls
#define TSRMLS_CC   , tsrm_ls
```

它的用法是第一个检索到storage的函数把它的指针传递给了下面的函数，参数是tsrm_ls，后面的函数直接根据接收的参数使用获取再传给其它函数，当然也可以不传，那样的话就得重新调用ts_resource()获取了。现在我们再看下EG宏展开的结果：

```
# define EG(v) TSRMG(executor_globals_id, zend_executor_globals *, v)

#define TSRMG(id, type, element)    (((type) (*((void ***) tsrm_ls))[TSRM_UNSHUFFLE_RSRC_ID(id)])->element)
```

比如： `EG(function_table) => (((zend_executor_globals *) (*((void ***) tsrm_ls))[executor_globals_id-1])->function_table)` ，这样我们在传了tsrm_ls的函数中就可能读取内存使用了。

PHP5的这种处理方式简单但是很不优雅，不管你用不用TSRM都不得不在函数中加上那两个宏，而且很容易遗漏。后来Anatol Belski在PHP的rfc提交了一种新的处理方式：https://wiki.php.net/rfc/native-tls，新的处理方式最终在PHP7版本得以实

现，通过静态TLS将各线程的storage保存在全局变量中，各函数中使用时直接读取即可。

linux下这种全局变量通过加上 `__thread` 定义，这样各线程更新这个变量就不会冲突了，实际这是gcc提供的，详细的内容这里不再展开，有兴趣的可以再查下详细的资料。举个例子：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

__thread int num = 0;

void* worker(void* arg){
    while(1){
        printf("thread:%d\n", num);
        sleep(1);
    }
}

int main(void)
{
    pthread_t tid;
    int ret;

    if ((ret = pthread_create(&tid, NULL, worker, NULL)) != 0){
        return 1;
    }

    while(1){
        num = 4;
        printf("main:%d\n", num);
        sleep(1);
    }

    return 0;
}
```

这个例子有两个线程，其中主线程修改了全局变量num，但是并没有影响另外一个线程。

PHP7中用于缓存各线程storage的全局变量定义在 `Zend/zend.c` :

```
#ifdef ZTS
//这些都是全局变量
ZEND_API int compiler_globals_id;
ZEND_API int executor_globals_id;
static HashTable *global_function_table = NULL;
static HashTable *global_class_table = NULL;
static HashTable *global_constants_table = NULL;
static HashTable *global_auto_globals_table = NULL;
static HashTable *global_persistent_list = NULL;
ZEND_TSRMLS_CACHE_DEFINE() //=>TSRM_TLS void *TSRMLS_CACHE = NULL; 展开后： __thread void *_tsrm_ls_cache = NULL; _tsrm_ls_cache就是各线程storage的地址
#endif
```

比如EG：

```
# define EG(v) ZEND_TSRMG(executor_globals_id, zend_executor_globals *, v)

#define ZEND_TSRMG TSRMG_STATIC
#define TSRMG_STATIC(id, type, element) (TSRMG_BULK_STATIC(id, type)->element)
#define TSRMG_BULK_STATIC(id, type) ((type) (*((void ***) TSRMLS_CACHE))[TSRM_UNSHUFFLE_RSRC_ID(id)])
```

EG(xxx)最终展开：((zend_executor_globals *) (((void *) _tsrm_ls_cache))[executor_globals_id-1]->xxx)。

7.1 概述

扩展是PHP的重要组成部分，它是PHP提供给开发者用于扩展PHP语言功能的主要方式。开发者可以用C/C++定义自己的功能，通过扩展嵌入到PHP中，灵活的扩展能力使得PHP拥有了大量、丰富的第三方组件，这些扩展很好的补充了PHP的功能、特性，使得PHP在web开发中得以大展身手。ext目录下有一个standard扩展，这个扩展提供了大量被大家所熟知的PHP函数：sleep()、usleep()、htmlspecialchars()、md5()、strtoupper()、substr()、array_merge()等等。

C语言是PHP之母，作为世界上非常优秀的一门语言，自它诞生至今，C语言早就了大量优秀、知名的项目：Linux、Nginx、MySQL、PHP、Redis、Memcached等等，感谢里奇带给这个世界如此伟大的一份礼物。C语言的优秀也折射到PHP身上，但是PHP内核提供的功能终究有限，如果你发现PHP在某些方面已经满足不了你的需求了，那么不妨试试扩展。

常见的，扩展可以在以下几个方面有所作为：

- 介入**PHP**的编译、执行阶段： 可以介入PHP框架执行的那5个阶段，比如opcache，就是重定义了编译函数
- 提供内部函数： 可以定义内部函数扩充PHP的函数功能，比如array、date等操作
- 提供内部类
- 实现**RPC**客户端： 实现与外部服务的交互，比如redis、mysql等
- 提升执行性能： PHP是解析型语言，在性能方面远不及C语言，可以将耗cpu的操作以C语言代替
- ......

当然扩展也不是万能，它只允许我们在PHP提供的框架之上进行一些特定的处理，同时限于SAPI的差异，扩展也必须要考虑到不同SAPI的实现特点。

PHP中的扩展分为两类：PHP扩展、Zend扩展，对内核而言这两个分别称之为：模块(module)、扩展(extension)，本章主要介绍是PHP扩展，也就是模块。

7.2 扩展的实现原理

PHP中扩展通过 `zend_module_entry` 这个结构来表示，此结构定义了扩展的全部信息：扩展名、扩展版本、扩展提供的函数列表以及PHP四个执行阶段的hook函数等，每一个扩展都需要定义一个此结构的变量，而且这个变量的名称格式必须是： `{module_name}_module_entry` ，内核正是通过这个结构获取到扩展提供的功能的。

扩展可以在编译PHP时一起编译(静态编译)，也可以单独编译为动态库，动态库需要加入到php.ini配置中去，然后在 `php_module_startup()` 阶段把这些动态库加载到PHP中：

```
int php_module_startup(sapi_module_struct *sf, zend_module_entry
 *additional_modules, uint num_additional_modules)
{
    ...
    //根据php.ini注册扩展
    php_ini_register_extensions();
    zend_startup_modules();

    zend_startup_extensions();
    ...
}
```

动态库就是在 `php_ini_register_extensions()` 这个函数中完成的注册：

```
//main/php_ini.c
void php_ini_register_extensions(void)
{
    //注册zend扩展
    zend_llist_apply(&extension_lists.engine, php_load_zend_exte
nsion_cb);
    //注册php扩展
    zend_llist_apply(&extension_lists.functions, php_load_php_ex
tension_cb);

    zend_llist_destroy(&extension_lists.engine);
    zend_llist_destroy(&extension_lists.functions);
}
```

extension_lists是一个链表，保存着根据`php.ini`中定义的`extension=xxx.so`取到的全部扩展名称，其中engine是zend扩展，functions为php扩展，依次遍历这两个数组然后调用`php_load_php_extension_cb()`或`php_load_zend_extension_cb()`进行各个扩展的加载：

```
static void php_load_php_extension_cb(void *arg)
{
#ifdef HAVE_LIBDL
    php_load_extension(*((char **) arg), MODULE_PERSISTENT, 0);
#endif
}
```

`HAVE_LIBDL`这个宏根据`dlopen()`函数是否存在设置的：

```
#Zend/Zend.m4
AC_DEFUN([LIBZEND_LIBDL_CHECKS],[
AC_CHECK_LIB(dl, dlopen, [LIBS="-ldl $LIBS"])
AC_CHECK_FUNC(dlopen,[AC_DEFINE(HAVE_LIBDL, 1,[ ])])
])
```

接着就是最关键的操作了，`php_load_extension()`：

```
//ext/standard/dl.c
PHPAPI int php_load_extension(char *filename, int type, int star
t_now)
{
    void *handle;
    char *libpath;
    zend_module_entry *module_entry;
    zend_module_entry *(*get_module)(void);
    ...
    //调用dlopen打开指定的动态连接库文件：xx.so
    handle = DL_LOAD(libpath);
    ...
    //调用dlsym获取get_module的函数指针
    get_module = (zend_module_entry *(*)(void)) DL_FETCH_SYMBOL(
handle, "get_module");
    ...
    //调用扩展的get_module()函数
    module_entry = get_module();
    ...
    //检查扩展使用的zend api是否与当前php版本一致
    if (module_entry->zend_api != ZEND_MODULE_API_NO) {
        DL_UNLOAD(handle);
        return FAILURE;
    }
    ...
    module_entry->type = type;
    //为扩展编号
    module_entry->module_number = zend_next_free_module();
    module_entry->handle = handle;

    if ((module_entry = zend_register_module_ex(module_entry)) =
= NULL) {
        DL_UNLOAD(handle);
        return FAILURE;
    }
    ...
}
```

`DL_LOAD()`、`DL_FETCH_SYMBOL()`这两个宏在linux下展开后就是：dlopen()、dlsym()，所以上面过程的实现就比较直观了：

- (1)dlopen()打开so库文件；
- (2)dlsym()获取动态库中`get_module()`函数的地址，`get_module()`是每个扩展都必须提供的一个接口，用于返回扩展`zend_module_entry`结构的地址;
- (3)调用扩展的`get_module()`，获取扩展的`zend_module_entry`结构;
- (4)zend api版本号检查，比如php7的扩展在php5下是无法使用的；
- (5)注册扩展，将扩展添加到`module_registry`中，这是一个全局HashTable，用于全部扩展的zend_module_entry结构；
- (6)如果扩展提供了内部函数则将这些函数注册到EG(function_table)中。

完成扩展的注册后，PHP将在不同的执行阶段依次调用每个扩展注册的当前阶段的hook函数。

7.3 扩展的构成及编译

7.3.1 扩展的构成

扩展首先需要创建一个`zend_module_entry`结构，这个变量必须是全局变量，且变量名必须是：`扩展名称_module_entry`，内核通过这个结构得到这个扩展都提供了哪些功能，换句话说，一个扩展可以只包含一个`zend_module_entry`结构，相当于定义了一个什么功能都没有的扩展。

```
//zend_modules.h
struct _zend_module_entry {
    unsigned short size; //sizeof(zend_module_entry)
    unsigned int zend_api; //ZEND_MODULE_API_NO
    unsigned char zend_debug; //是否开启debug
    unsigned char zts; //是否开启线程安全
    const struct _zend_ini_entry *ini_entry;
    const struct _zend_module_dep *deps;
    const char *name; //扩展名称，不能重复
    const struct _zend_function_entry *functions; //扩展提供的内部函数列表
    int (*module_startup_func)(INIT_FUNC_ARGS); //扩展初始化回调函数，PHP_MINIT_FUNCTION或ZEND_MINIT_FUNCTION定义的函数
    int (*module_shutdown_func)(SHUTDOWN_FUNC_ARGS); //扩展关闭时回调函数
    int (*request_startup_func)(INIT_FUNC_ARGS); //请求开始前回调函数

    int (*request_shutdown_func)(SHUTDOWN_FUNC_ARGS); //请求结束时回调函数
    void (*info_func)(ZEND_MODULE_INFO_FUNC_ARGS); //php_info展示的扩展信息处理函数
    const char *version; //版本
    ...
    unsigned char type;
    void *handle;
    int module_number; //扩展的唯一编号
    const char *build_id;
};
```

这个结构包含很多成员，但并不是所有的都需要自己定义，经常用到的主要有下面几个：

- **name:** 扩展名称，不能重复
- **functions:** 扩展定义的内部函数entry
- **module_startup_func:** PHP在模块初始化时回调的hook函数，可以使扩展介入module startup阶段
- **module_shutdown_func:** 在模块关闭阶段回调的函数
- **request_startup_func:** 在请求初始化阶段回调的函数

- **request_shutdown_func:** 在请求结束阶段回调的函数
- **info_func:** php_info()函数时调用，用于展示一些配置、运行信息
- **version:** 扩展版本

除了上面这些需要手动设置的成员，其它部分可以通过`STANDARD_MODULE_HEADER`、`STANDARD_MODULE_PROPERTIES`宏统一设置，扩展提供的内部函数及四个执行阶段的钩子函数是扩展最常用到的部分，几乎所有的扩展都是基于这两部分实现的。有了这个结构还需要提供一个接口来获取这个结构变量，这个接口是统一的，扩展中通过`ZEND_GET_MODULE(extension_name)`完成这个接口的定义:

```
//zend_API.h
#define ZEND_GET_MODULE(name) \
    BEGIN_EXTERN_C()\
    ZEND_DLEXPORT zend_module_entry *get_module(void) { return &
name##_module_entry; }\
    END_EXTERN_C()
```

展开后可以看到，实际就是定义了一个get_module()函数，返回扩展zend_module_entry结构的地址，这就是为什么这个结构的变量名必须是`扩展名称_module_entry`这种格式的原因。

有了扩展的zend_module_entry结构以及获取这个结构的接口一个合格的扩展就编写完成了，只是这个扩展目前还什么都干不了：

```
#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"

zend_module_entry mytest_module_entry = {
    STANDARD_MODULE_HEADER,
    "mytest",
    NULL, //mytest_functions,
    NULL, //PHP_MINIT(mytest),
    NULL, //PHP_MSHUTDOWN(mytest),
    NULL, //PHP_RINIT(mytest),
    NULL, //PHP_RSHUTDOWN(mytest),
    NULL, //PHP_MINFO(mytest),
    "1.0.0",
    STANDARD_MODULE_PROPERTIES
};

ZEND_GET_MODULE(mytest)
```

编译、安装后执行`php -m`就可以看到my_test这个扩展了。

7.3.2 编译工具

PHP提供了几个脚本工具用于简化扩展的实现：ext_skel、phpize、php-config，后面两个脚本主要配合autoconf、automake生成Makefile。在介绍这几个工具之前，我们先看下PHP安装后的目录结构，因为很多脚本、配置都放置在安装后的目录中，比如PHP的安装路径为：/usr/local/php7，则此目录的主要结构：

```
|---php7
|   |---bin //php编译生成的二进制程序目录
|       |---php //cli模式的php
|       |---phpize
|       |---php-config
|       |---...
|   |---etc     //一些sapi的配置
|   |---include //php源码的头文件
|       |---php
|           |---main    //PHP中的头文件
|           |---Zend    //Zend头文件
|           |---TSRM    //TSRM头文件
|           |---ext     //扩展头文件
|           |---sapi    //SAPI头文件
|           |---include
|   |---lib //依赖的so库
|       |---php
|           |---extensions  //扩展so保存目录
|           |---build       //编译时的工具、m4配置等，编写扩展是会用到
|               |---acinclude.m4    //PHP自定义的autoconf宏
|               |---libtool.m4      //libtool定义的autoconf宏，aci
nclude.m4、libtool.m4会被合成aclocal.m4
|               |---phpize.m4       //PHP核心configure.in配置
|               |---...
|           |---...
|   |---php
|   |---sbin //SAPI编译生成的二进制程序，php-fpm会放在这
|   |---var  //log、run日志
```

7.3.2.1 ext_skel

这个脚本位于PHP源码/ext目录下，它的作用是用来生成扩展的基本骨架，帮助开发者快速生成一个规范的扩展结构，可以通过以下命令生成一个扩展结构：

```
./ext_skel --extname=扩展名称
```

执行完以后会在ext目录下新生成一个扩展目录，比如extname是mytest，则将生成以下文件：

```
|---mytest
|    |---config.m4      //autoconf规则的编译配置文件
|    |---config.w32     //windows环境的配置
|    |---CREDITS
|    |---EXPERIMENTAL
|    |---include        //依赖库的include头文件，可以不用
|    |---mytest.c       //扩展源码
|    |---php_mytest.h   //头文件
|    |---mytest.php     //用于在PHP中测试扩展是否可用，可以不用
|    |---tests          //测试用例，执行make test时将执行、验证这些用例
|        |---001.phpt
```

这个脚本主要生成了编译需要的配置以及扩展的基本结构，初步生成的这个扩展可以成功的编译、安装、使用，实际开发中我们可以使用这个脚本生成一个基本结构，然后根据具体的需要逐步完善。

7.3.2.2 php-config

这个脚本为PHP源码中的/script/php-config.in，PHP安装后被移到安装路径的/bin目录下，并重命名为php-config，这个脚本主要是获取PHP的安装信息的，主要有：

- **PHP**安装路径
- **PHP**版本
- **PHP**源码的头文件目录： main、Zend、ext、TSRM中的头文件，编写扩展时会用到这些头文件，这些头文件保存在PHP安装位置/include/php目录下
- **LDFLAGS**： 外部库路径，比如： `-L/usr/bib -L/usr/local/lib`
- 依赖的外部库 ： 告诉编译器要链接哪些文件， `-lcrypt -lresolv -lcrypt` 等等
- 扩展存放目录 ： 扩展.so保存位置，安装扩展make install时将安装到此路径下
- 编译的**SAPI**： 如cli、fpm、cgi等
- **PHP**编译参数 ： 执行./configure时带的参数
- ...

这个脚本在编译扩展时会用到，执行 `./configure --with-php-config=xxx` 生成Makefile时作为参数传入即可，它的作用是提供给configure.in获取上面几个配置，生成Makefile。

7.3.2.3 phpize

这个脚本主要是操作复杂的autoconf/automake/autoheader/autolocal等系列命令，用于生成configure文件，GNU auto系列的工具众多，这里简单介绍下基本的使用：

(1)autoscan： 在源码目录下扫描，生成configure.scan，然后把这个文件重名为为configure.in，可以在这个文件里对依赖的文件、库进行检查以及配置一些编译参数等。

(2)aclocal： automake中有很多宏可以在configure.in或其它.m4配置中使用，这些宏必须定义在aclocal.m4中，否则将无法被autoconf识别，aclocal可以根据configure.in自动生成aclocal.m4，另外，autoconf提供的特性不可能满足所有的需求，所以autoconf还支持自定义宏，用户可以在acinclude.m4中定义自己的宏，然后在执行aclocal生成aclocal.m4时也会将acinclude.m4加载进去。

(3)autoheader： 它可以根据configure.in、aclocal.m4生成一个C语言"define"声明的头文件模板(config.h.in)供configure执行时使用，比如很多程序会通过configure提供一些enable/disable的参数，然后根据不同的参数决定是否开启某些选项，这种就可以根据编译参数的值生成一个define宏，比如： `--enabled-xxx` 生成 `#define ENABLED_XXX 1` ，否则默认生成 `#define ENABLED_XXX 0` ，代码里直接使用这个宏即可。比如configure.in文件内容如下：

```
AC_PREREQ([2.63])
AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])

AC_CONFIG_HEADERS([config.h])

AC_ARG_ENABLE(xxx, "--enable-xxx if enable xxx",[
    AC_DEFINE([ENABLED_XXX], [1], [enabled xxx])
],
[
    AC_DEFINE([ENABLED_XXX], [0], [disabled xxx])
])

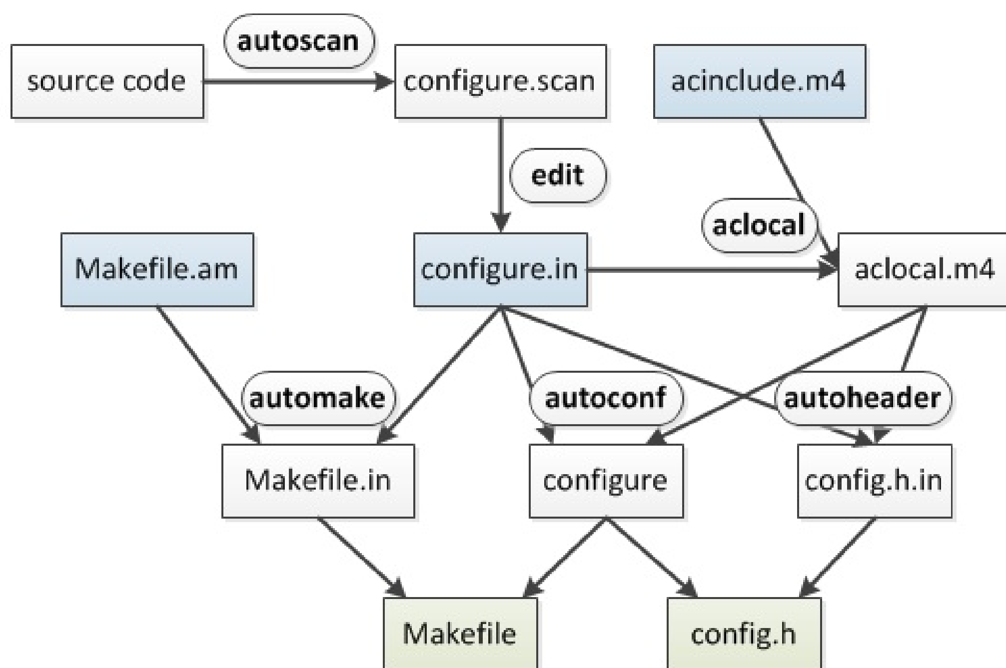
AC_OUTPUT
```

执行autoheader后将生成一个config.h.in的文件，里面包含 `#undef ENABLED_XXX` ，最终执行 `./configure --enable-xxx` 后将生成一个config.h文件，包含 `#define ENABLED_XXX 1` 。

(4)autoconf： 将configure.in中的宏展开生成configure、config.h，此过程会用到aclocal.m4中定义的宏。

(5)automake： 将Makefile.am中定义的结构建立Makefile.in，然后configure脚本将生成的Makefile.in文件转换为Makefile。

各步骤之间的转化关系如下图：



编写PHP扩展时并不需要操作上面全部的步骤，PHP提供了两个编辑好的配置：configure.in、acinclude.m4，这两个配置是从PHP安装路径/lib/php/build目录下的phpize.m4、acinclude.m4复制生成的，其中configure.in中定义了一些PHP内核相关的配置检查项，另外这个文件会include每个扩展各自的配置:config.m4，所以编写扩展时我们只需要在config.m4中定义扩展自己的配置就可以了，不需要关心依赖的PHP内核相关的配置，在扩展所在目录下执行phpize就可以生成扩展的configure、config.h文件了。

configure.in(phpize.m4)：

```
AC_PREREQ(2.59)
AC_INIT(config.m4)
...
#--with-php-config参数
PHP_ARG_WITH(php-config,,
[  --with-php-config=PATH  Path to php-config [php-config]], php
-config, no)

PHP_CONFIG=$PHP_PHP_CONFIG
...
#加载扩展配置
sinclude(config.m4)
...
AC_CONFIG_HEADER(config.h)

AC_OUTPUT()
```

phpize中的主要操作：

(1)phpize_check_configm4: 检查扩展的config.m4是否存在。

(2)phpize_check_build_files: 检查php安装路径下的lib/php/build/，这个目录下包含PHP自定义的autoconf宏文件acinclude.m4以及libtool；检查扩展所在目录。

(3)phpize_print_api_numbers: 输出PHP Api Version、Zend Module Api No、Zend Extension Api No信息。

```
phpize_get_api_numbers()
{
  # extracting API NOs:
  PHP_API_VERSION=`grep '#define PHP_API_VERSION' $includedir/ma
in/php.h|$SED 's/#define PHP_API_VERSION//'`
  ZEND_MODULE_API_NO=`grep '#define ZEND_MODULE_API_NO' $include
dir/Zend/zend_modules.h|$SED 's/#define ZEND_MODULE_API_NO//'`
  ZEND_EXTENSION_API_NO=`grep '#define ZEND_EXTENSION_API_NO' $i
ncludedir/Zend/zend_extensions.h|$SED 's/#define ZEND_EXTENSION_
API_NO//'`
}
```

(4)phpize_copy_files: 将PHP安装位置/lib/php/build目录下的mkdep.awk scan_makefile_in.awk shtool libtool.m4四个文件拷到扩展的build目录下，然后将acinclude.m4 Makefile.global config.sub config.guess ltmain.sh run-tests*.php文件拷到扩展根目录，最后将acinclude.m4、build/libtool.m4合并到扩展目录下的aclocal.m4文件中。

```
phpize_copy_files()
{
  test -d build || mkdir build

  (cd "$phpdir" && cp $FILES_BUILD "$builddir"/build)
  (cd "$phpdir" && cp $FILES "$builddir")
  #acinclude.m4、libtool.m4合并到aclocal.m4
  (cd "$builddir" && cat acinclude.m4 ./build/libtool.m4 > acloc
al.m4)
}
```

(5)phpize_replace_prefix: 将PHP安装位置/lib/php/build/phpize.m4拷贝到扩展目录下，将文件中的prefix替换为PHP安装路径，然后重命名为configure.in。

```
phpize_replace_prefix()
{
  $SED \
  -e "s#/usr/local/php7#$prefix#" \
  < "$phpdir/phpize.m4" > configure.in
}
```

(6)phpize_check_shtool: 检查/build/shtool。

(7)phpize_check_autotools: 检查autoconf、autoheader。

(8)phpize_autotools 执行autoconf生成configure，然后再执行autoheader生成config.h。

7.3.3 编写扩展的基本步骤

编写一个PHP扩展主要分为以下几步：

- 通过ext目录下ext_skel脚本生成扩展的基本框架：`./ext_skel --extname`；
- 修改config.m4配置：设置编译配置参数、设置扩展的源文件、依赖库/函数检查等等；
- 编写扩展要实现的功能：按照PHP扩展的格式以及PHP提供的API编写功能；
- 生成configure：扩展编写完成后执行phpize脚本生成configure及其它配置文件；
- 编译&安装：./configure、make、make install，然后将扩展的.so路径添加到php.ini中。

最后就可以在PHP中使用这个扩展了。

7.3.4 config.m4

config.m4是扩展的编译配置文件，它被include到configure.in文件中，最终被autoconf编译为configure，编写扩展时我们只需要在config.m4中修改配置即可，一个简单的扩展配置只需要包含以下内容：

```
PHP_ARG_WITH(扩展名称, for mytest support,
Make sure that the comment is aligned:
[  --with-扩展名称             Include xxx support])

if test "$PHP_扩展名称" != "no"; then
    PHP_NEW_EXTENSION(扩展名称, 源码文件列表, $ext_shared,, -DZEND_
ENABLE_STATIC_TSRMLS_CACHE=1)
fi
```

PHP在acinclude.m4中基于autoconf/automake的宏封装了很多可以直接使用的宏，下面介绍几个比较常用的宏：

(1)PHP_ARG_WITH(arg_name,check message,help info): 定义一个`--with-feature[=arg]`这样的编译参数，调用的是autoconf的*ACARG_WITH*，这个宏有*5*个参数，常用的是前三个，分别表示：参数名、执行*./configure*是展示信息、执行*--help*时展示信息，第*4*个参数为默认值，如果不定义默认为*"no"*，通过这个宏定义的参数可以在*config.m4*中通过*`$PHP*参数名*(*大写*)`*访问，比如：

```
PHP_ARG_WITH(aaa, aaa-configure, help aa)

#后面通过$PHP_AAA就可以读取到--with-aaa=xxx设置的值了
```

(2)PHP_ARG_ENABLE(arg_name,check message,help info): 定义一个 `--enable-feature[=arg]` 或 `--disable-feature` 参数， `--disable-feature` 等价于 `--enable-feature=no` ，这个宏与PHP_ARG_WITH类似，通常情况下如果配置的参数需要额外的arg值会使用PHP_ARG_WITH，而如果不需要arg值，只用于开关配置则会使用PHP_ARG_ENABLE。

(3)AC_MSG_CHECKING()/AC_MSG_RESULT()/AC_MSG_ERROR(): ./configure时输出结果，其中error将会中断configure执行。

(4)AC_DEFINE(variable, value, [description]): 定义一个宏，比如： `AC_DEFINE(IS_DEBUG, 1, [])` ，执行autoheader时将在头文件中生成： `#define IS_DEBUG 1` 。

(5)PHP_ADD_INCLUDE(path): 添加include路径，即： `gcc -Iinclude_dir` ， `#include "file";` 将先在通过-I指定的目录下查找，扩展引用了外部库或者扩展下分了多个目录的情况下会用到这个宏。

(6)PHP_CHECK_LIBRARY(library, function [, action-found [, action-not-found [, extra-libs]]]): 检查依赖的库中是否存在需要的function，action-found为存在时执行的动作，action-not-found为不存在时执行的动作，比如扩展里使用到线程pthread，检查pthread_create()，如果没找到则终止./configure执行：

```
PHP_ADD_INCLUDE(pthread, pthread_create, [], [
    AC_MSG_ERROR([not find pthread_create() in lib pthread])
])
```

(7)AC_CHECK_FUNC(function, [action-if-found], [action-if-not-found]): 检查函数是否存在。 **(8)PHP_ADD_LIBRARY_WITH_PATH($LIBNAME, $XXX_DIR/$PHP_LIBDIR, XXX_SHARED_LIBADD):** 添加链接库。

(9)PHP_NEW_EXTENSION(extname, sources [, shared [, sapi_class [, extra-cflags [, cxx [, zend_ext]]]]]): 注册一个扩展，添加扩展源文件，确定此扩展是动态库还是静态库，每个扩展的config.m4中都需要通过这个宏完成扩展的编译配置。

更多autoconf及PHP封装的宏大家可以在用到的时候再自行检索，同时ext目录下有大量的示例可供参考。

7.4 钩子函数

PHP为扩展提供了5个钩子函数，PHP执行到不同阶段时回调各个扩展定义的钩子函数，扩展可以通过这些钩子函数介入到PHP生命周期的不同阶段中去，这些钩子函数的定义非常简单，PHP提供了对应的宏，定义完成后只需要设置 `zend_module_entry` 对应的函数指针即可。

前面已经介绍过PHP生命周期的几个阶段，这几个钩子函数执行的先后顺序：module startup -> request startup -> 编译、执行 -> request shutdown -> post deactivate -> module shutdown。

7.4.1 module_startup_func

这个函数在PHP模块初始化阶段执行，通常情况下，此过程只会在SAPI启动后执行一次。这个阶段可以进行内部类的注册，如果你的扩展提供了类就可以在此函数中完成注册；除了类还可以在此函数中注册扩展定义的常量；另外，扩展可以在此阶段覆盖PHP编译、执行的两个函数指针：zend_compile_file、zend_execute_ex，从而可以接管PHP的编译、执行，opcache的实现原理就是替换了zend_compile_file，从而使得PHP编译时调用的是opcache自己定义的编译函数，对编译后的结果进行缓存。

此钩子函数通过 `PHP_MINIT_FUNCTION()` 或 `ZEND_MINIT_FUNCTION()` 宏完成定义：

```
PHP_MINIT_FUNCTION(extension_name)
{
    ...
}
```

展开后：

```
zm_startup_extension_name(int type, int module_number)
{
    ...
}
```

最后通过 `PHP_MINIT()` 或 `ZEND_MINIT()` 宏将zend_module_entry的module_startup_func设置为上面定义的函数。

```
#define PHP_MINIT          ZEND_MODULE_STARTUP_N
#define ZEND_MINIT          ZEND_MODULE_STARTUP_N

#define ZEND_MODULE_STARTUP_N(module)       zm_startup_##module
```

7.4.2 request_startup_func

此函数在编译、执行之前回调，fpm模式下每一个http请求就是一个request，脚本执行前将首先执行这个函数。如果你的扩展需要针对每一个请求进行处理则可以设置这个函数，如：对请求进行filter、根据请求ip获取所在城市、对请求/返回数据加解密等。此函数通过 `PHP_RINIT_FUNCTION()` 或 `ZEND_RINIT_FUNCTION()` 宏定义：

```
PHP_RINIT_FUNCTION(extension_name)
{
    ...
}
```

展开后：

```
zm_activate_extension_name(int type, int module_number)
{
    ...
}
```

获取函数地址的宏： `PHP_RINIT()` 或 `ZEND_RINIT()` ：

```
#define PHP_RINIT          ZEND_MODULE_ACTIVATE_N
#define ZEND_RINIT          ZEND_MODULE_ACTIVATE_N

#define ZEND_MODULE_ACTIVATE_N(module)      zm_activate_##module
```

7.4.3 request_shutdown_func

此函数在请求结束时被调用，通过 `PHP_RSHUTDOWN_FUNCTION()` 或 `ZEND_RSHUTDOWN_FUNCTION()` 宏定义：

```
PHP_RSHUTDOWN_FUNCTION(extension_name)
{
    ...
}
```

函数地址通过 `PHP_RSHUTDOWN()` 或 `ZEND_RSHUTDOWN()` 获取：

```
#define PHP_RSHUTDOWN      ZEND_MODULE_DEACTIVATE_N
#define ZEND_RSHUTDOWN      ZEND_MODULE_DEACTIVATE_N

#define ZEND_MODULE_DEACTIVATE_N(module)    zm_deactivate_##modu
le
```

7.4.4 post_deactivate_func

这个函数比较特殊，一般很少会用到，实际它也是在请求结束之后调用的，它比request_shutdown_func更晚执行：

```
void php_request_shutdown(void *dummy)
{
    ...
    //调用各扩展的request_shutdown_func
    if (PG(modules_activated)) {
        zend_deactivate_modules();
    }
    //关闭输出:发送http header
    php_output_deactivate();

    //释放超全局变量：$_GET、$_POST...
    ...
    //关闭编译器、执行器
    zend_deactivate();

    //调用每个扩展的post_deactivate_func
    zend_post_deactivate_modules();
    ...
}
```

从上面的执行顺序可以看出，request_shutdown_func、post_deactivate_func是先后执行的，此函数通过 `ZEND_MODULE_POST_ZEND_DEACTIVATE_D()` 宏定义， `ZEND_MODULE_POST_ZEND_DEACTIVATE_N()` 获取函数地址：

```
#define ZEND_MINIT          ZEND_MODULE_STARTUP_N
#define ZEND_MODULE_POST_ZEND_DEACTIVATE_N(module)  zm_post_zend
_deactivate_##module
```

7.4.5 module_shutdown_func

模块关闭阶段回调的函数，与module_startup_func对应，此阶段主要可以进行一些资源的清理，通过 `PHP_MSHUTDOWN_FUNCTION()` 或 `ZEND_MSHUTDOWN_FUNCTION()` 定义：

```
PHP_MSHUTDOWN_FUNCTION(extension_name)
{
    ...
}
```

通过 `PHP_MSHUTDOWN()` 或 `ZEND_MSHUTDOWN()` 获取函数地址：

```
#define PHP_MSHUTDOWN      ZEND_MODULE_SHUTDOWN_N
#define ZEND_MSHUTDOWN      ZEND_MODULE_SHUTDOWN_N

#define ZEND_MODULE_SHUTDOWN_N(module)      zm_shutdown_##module
```

7.4.6 小节 上面详细介绍了各个阶段定义的钩子函数的格式，使用gdb调试扩展时可以根据展开后实际的函数名称设置断点。这些钩子实际已经为扩展构造了一个整体的框架，通过这几个钩子扩展已经能实现很多功能了，后面我们介绍的很多内容都是在这几个函数中完成的，比如内部类的注册、常量注册、资源注册等。如果扩展名称为mytest，则最终定义的扩展：

```
PHP_MINIT_FUNCTION(mytest)
{
    ...
}

PHP_RINIT_FUNCTION(mytest)
{
    ...
}

PHP_RSHUTDOWN_FUNCTION(mytest)
{
    ...
}

PHP_MSHUTDOWN_FUNCTION(mytest)
{
    ...
}

zend_module_entry mytest_module_entry = {
    STANDARD_MODULE_HEADER,
    "mytest",
    NULL, //mytest_functions,
    PHP_MINIT(mytest),
    PHP_MSHUTDOWN(mytest),
    PHP_RINIT(mytest),
    PHP_RSHUTDOWN(mytest),
    NULL, //PHP_MINFO(mytest),
    "1.0.0",
    STANDARD_MODULE_PROPERTIES
};

ZEND_GET_MODULE(mytest)
```

7.5 运行时配置

7.5.1 全局变量(资源)

使用C语言开发程序时经常会使用全局变量进行数据存储，这就涉及前面已经介绍过的一个问题：线程安全，PHP设计了TSRM（即：线程安全资源管理器）用于解决这个问题，内核中频繁使用到的EG、CG等都是根据是否开启ZTS封装的宏，同样的，在扩展中也需要必须按照TSRM的规范定义全局变量，除非你的扩展不支持多线程的环境。

PHP为扩展的全局变量提供了一种存储方式：每个扩展将自己所有的全局变量统一定义在一个结构体中，然后将这个结构体注册到TSRM中，这样扩展就可以像使用EG、CG那样访问这个结构体。

这个结构体的定义通过 `ZEND_BEGIN_MODULE_GLOBALS(extension_name)` 、 `ZEND_END_MODULE_GLOBALS(extension_name)` 两个宏完成，这两个宏必须成对出现，中间定义扩展需要的全局变量即可。

```
ZEND_BEGIN_MODULE_GLOBALS(mytest)
    zend_long     open_cache;
    HashTable     class_table;
ZEND_END_MODULE_GLOBALS(mytest)
```

展开后实际就是个普通的struct：

```
typedef struct _zend_mytest_globals {
    zend_long    open_cache;
    HashTable    class_table;
}zend_mytest_globals;
```

接着创建一个此结构体的全局变量，这时候就会涉及ZTS了，如果未开启线程安全直接创建普通的全局变量即可，如果开启线程安全了则需要向TSRM注册，得到一个唯一的资源id，这个操作也由专门的宏来完成： `ZEND_DECLARE_MODULE_GLOBALS(extension_name)` ，展开后：

```
//ZTS：此时只是定义资源id，并没有向TSRM注册
ts_rsrc_id mytest_globals_id;

//非ZTS
zend_mytest_globals mytest_globals;
```

最后需要定义一个像EG、CG那样的宏用于访问扩展的全局资源结构体，这一步将使用 `ZEND_MODULE_GLOBALS_ACCESSOR()` 宏完成：

```
#define MYTEST_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(mytest, v)
```

看起来是不是跟EG、CG的定义非常像？这个宏展开后：

```
//ZTS
#define MYTEST_G(v) ZEND_TSRMG(mytest_globals_id, zend_mytest_gl
obals *, v)

//非ZTS
#define MYTEST_G(v) (mytest_globals.v)
```

接下来就可以在扩展中通过：MYTEST_G(opene_cache)、MYTEST_G(class_table)对结构体成员进行读写了。通常会把这个全局资源结构体及结构体的访问宏定义在头文件中，然后把全局变量的声明放到源文件中：

```
//php_mytest.h
#define MYTEST_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(mytest, v)

ZEND_BEGIN_MODULE_GLOBALS(mytest)
    zend_long   open_cache;
    HashTable   class_table;
ZEND_END_MODULE_GLOBALS(mytest)

//mytest.c
ZEND_DECLARE_MODULE_GLOBALS(mytest)
```

> 在一个扩展中并不是只能定义一个全局变量结构，数目是不限制的。

7.5.2 php.ini配置

php.ini是PHP主要的配置文件，解析时PHP将在这些地方依次查找该文件：当前工作目录、环境变量PHPRC指定目录、编译时指定的路径，在命令行模式下，php.ini的查找路径可以用 `-c` 参数替代。

该文件的语法非常简单： `配置标识符 = 值` 。空白字符和用分号';'开始的行被忽略，[xxx]行也被忽略；配置标识符大写敏感，通常会用'.'区分不同的节；值可以是数字、字符串、PHP常量、位运算表达式。

关于php.ini的解析过程本节不作介绍，只从应用的角度介绍如何在一个扩展中获取一个配置项，通常会把php.ini的配置映射到一个变量，从而在使用时直接读取那个变量，也就是把所有的配置转化为了C语言中的变量，扩展中一般会把php.ini配置映射到上一节介绍的全局变量(资源)，要想实现这个转化需要在扩展中为每一项配置设置映射规则：

```
PHP_INI_BEGIN()
    //每一项配置规则
    ...
PHP_INI_END();
```

这两个宏实际只是把各配置规则组成一个数组，配置规则通过 `STD_PHP_INI_ENTRY()` 设置：

```
STD_PHP_INI_ENTRY(name,default_value,modifiable,on_modify,proper
ty_name,struct_type,struct_ptr)
```

- **name:** php.ini中的配置标识符
- **default_value:** 默认值，注意不管转化后是什么类型，这里必须设置为字符串
- **modifiable:** 可修改等级，ZEND_INI_USER为可以在php脚本中修改，ZEND_INI_SYSTEM为可以在php.ini中修改，还有一个ZEND_INI_PERDIR，ZEND_INI_ALL表示三种都可以，通常情况下设置为ZEND_INI_ALL、ZEND_INI_SYSTEM即可
- **on_modify:** 函数指针，用于指定发现这个配置后赋值处理的函数，默认提供了5个：OnUpdateBool、OnUpdateLong、OnUpdateLongGEZero、OnUpdateReal、OnUpdateString、OnUpdateStringUnempty，支持可以自定义

* **property_name:** 要映射到的结构struct_type中的成员
* **struct_type:** 映射结构的类型
* **struct_ptr:** 映射结构的变量地址，发现配置后会

> 除了STD_PHP_INI_ENTRY()这个宏还有一个类似的宏 `STD_PHP_INI_BOOLEAN()` ，用法一致，差别在于后者会自动把配置添加到phpinfo()输出中。

这个宏展开后生成一个 `zend_ini_entry_def` 结构：

```
typedef struct _zend_ini_entry_def {
    const char *name;
    int (*on_modify)(zend_ini_entry *entry, zend_string *new_val
ue, void *mh_arg1, void *mh_arg2, void *mh_arg3, int stage);
    void *mh_arg1; //映射成员所在结构体的偏移:offsetof(type, member-
designator)取到
    void *mh_arg2; //要映射到结构的地址
    void *mh_arg3;
    const char *value;//默认值
    void (*displayer)(zend_ini_entry *ini_entry, int type);
    int modifiable;

    uint name_length;
    uint value_length;
} zend_ini_entry_def;
```

比如将php.ini中的 `mytest.opene_cache` 值映射到 `MYTEST_G()` 结构中的open_cache，类型为zend_long，默认值109，则可以这么定义：

```
PHP_INI_BEGIN()
    STD_PHP_INI_ENTRY("mytest.open_cache", "109", PHP_INI_ALL, O
nUpdateLong, open_cache, zend_mytest_globals, mytest_globals)
PHP_INI_END();
```

property_name设置的是要映射到的结构成员 `mytest_globals->open_cache` ，zend_mytest_globals、mytest_globals都是宏展开后的实际值，前者是结构体类型，后者是具体分配的变量，上面的定义展开后：

```
static const zend_ini_entry_def ini_entries[] = {
    {
        "mytest.open_cache",
        OnUpdateLong,
        (void *) XtOffsetOf(zend_mytest_globals, open_cache), //
获取成员在结构体中的内存偏移
        (void*)&mytest_globals,
        NULL,
        "109",
        NULL,
        PHP_INI_ALL,
        sizeof("mytest.open_cache")-1,
        sizeof("109")-1
    },
    { NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0, 0, 0}
}
```

> `XtOffsetOf()` 这个宏在linux环境下展开就是 `offsetof()` ，用来获取一个结构体成员的offset，比如：
>
> # include
>
> # include
>
> typedef struct{
> int id;
> char *name;
> }my_struct;
>
> int main(void)
> {
> printf("%d\n", (void*)offsetof(my_struct, name));
> return 0;
> }
>
> 通过这个offset及结构体指针就可以读取这个成员： `(char*)my_sutct + offset` ，等价于 `my_sutct->name` 。

定义完上面的配置映射规则后就可以进行映射了，这一步通过 `REGISTER_INI_ENTRIES()` 完成，这个宏展开后： `zend_register_ini_entries(ini_entries, module_number)` ，ini_entries是 `PHP_INI_BEGIN/END()` 两个宏生成的配置映射规则数组，通常会把这个操作放到 `PHP_MINIT_FUNCTION()` 中，注意：此时php.ini已经解析到 `configuration_hash` 哈希表中， `zend_register_ini_entries()` 将根据配置name查找这个哈希表，如果找到了表明用户在php.ini中配置了该项，然后将调用此规则指定的on_modify函数进行赋值，比如上面的示例将调用 `OnUpdateLong()` 处理，整体的流程：

```
ZEND_API int zend_register_ini_entries(const zend_ini_entry_def
*ini_entry, int module_number)
{
    zend_ini_entry *p;
    zval *default_value;
    HashTable *directives = registered_zend_ini_directives;

    while (ini_entry->name) {
        //分配zend_ini_entry结构
        p = pemalloc(sizeof(zend_ini_entry), 1);
        //zend_ini_entry初始化
        ...

        //添加到registered_zend_ini_directives，EG(ini_directives
)也是指向此HashTable
        if (zend_hash_add_ptr(directives, p->name, (void*)p) ==
NULL) {
            ...
        }

        //zend_get_configuration_directive()最终将调用cfg_get_entr
y()
        //从configuration_hash哈希表中查找配置，如果没有找到将使用默认值

        default_value = zend_get_configuration_directive(p->name
)
        ...
        if (p->on_modify) {
            //调用定义的赋值handler处理
            p->on_modify(p, p->value, p->mh_arg1, p->mh_arg2, p-
>mh_arg3, ZEND_INI_STAGE_STARTUP);
        }
    }
}
```

`OnUpdateLong()` 赋值处理:

```
ZEND_API ZEND_INI_MH(OnUpdateLong)
{
    zend_long *p;
#ifndef ZTS
    //存储结构的指针
    char *base = (char *) mh_arg2;
#else
    char *base;
    //ZTS下需要向TSRM中获取存储结构的指针
    base = (char *) ts_resource(*((int *) mh_arg2));
#endif
    //指向结构体成员的位置
    p = (zend_long *) (base+(size_t) mh_arg1);
    //将值转为zend_long
    *p = zend_atol(ZSTR_VAL(new_value), (int)ZSTR_LEN(new_value)
);
    return SUCCESS;
}
```

如果PHP提供的几个on_modify不能满足需求可以自定义on_modify函数，举个例子：将php.ini中的配置 `mytest.class` 插入MYTESY_G(class_table)哈希表，则可以在扩展中定义这样一个on_modify： `ZEND_INI_MH(OnUpdateAddArray)` ，将php.ini映射到全局变量的完整代码：

```
//php_mytest.h
#define MYTEST_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(mytest, v)

ZEND_BEGIN_MODULE_GLOBALS(mytest)
    zend_long    open_cache;
    HashTable    class_table;
ZEND_END_MODULE_GLOBALS(mytest)

//自定义on_modify函数
ZEND_API ZEND_INI_MH(OnUpdateAddArray);
```

```
//mytest.c
ZEND_DECLARE_MODULE_GLOBALS(mytest)
```

```
PHP_INI_BEGIN()
    STD_PHP_INI_ENTRY("mytest.open_cache", "109", PHP_INI_ALL, O
nUpdateLong, open_cache, zend_mytest_globals, mytest_globals)
    STD_PHP_INI_ENTRY("mytest.class", "stdClass", PHP_INI_ALL, O
nUpdateAddArray, class_table, zend_mytest_globals, mytest_global
s)
PHP_INI_END();

ZEND_API ZEND_INI_MH(OnUpdateAddArray)
{
    HashTable   *ht;
    zval    val;
#ifndef ZTS
    char *base = (char *) mh_arg2;
#else
    char *base;
    base = (char *) ts_resource(*((int *) mh_arg2));
#endif

    ht = (HashTable*)(base+(size_t) mh_arg1);
    ZVAL_NULL(&val);
    zend_hash_add(ht, new_value, &val);
}

PHP_MINIT_FUNCTION(mytest)
{
    zend_hash_init(&MYTEST_G(class_table), 0, NULL, NULL, 1);
    //将php.ini解析到指定结构体
    REGISTER_INI_ENTRIES();

    printf("open_cache %d\n", MYTEST_G(open_cache));
}

zend_module_entry mytest_module_entry = {
    STANDARD_MODULE_HEADER,
    "mytest",
    NULL,//mytest_functions,
    PHP_MINIT(mytest),
    NULL,//PHP_MSHUTDOWN(mytest),
```

```
    NULL,//PHP_RINIT(mytest),
    NULL,//PHP_RSHUTDOWN(mytest),
    NULL,//PHP_MINFO(mytest),
    "1.0.0",
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_TIMEOUT
#ifdef ZTS
ZEND_TSRMLS_CACHE_DEFINE()
#endif
ZEND_GET_MODULE(mytest)
#endif
```

本节主要介绍了如何将php.ini配置项解析到C语言变量中，总结下主要分为两步：

- 定义解析规则：通过PHP_INI_BEGIN()、PHP_INI_END()、STD_PHP_INI_ENTRY()配置
- 执行规则映射：由REGISTER_INI_ENTRIES()来完成，这个操作之后解析目的变量就可以使用了

7.6 函数

7.6.1 内部函数注册

通过扩展可以将C语言实现的函数提供给PHP脚本使用，如同大量PHP内置函数一样，这些函数统称为内部函数(internal function)，与PHP脚本中定义的用户函数不同，它们无需经历用户函数的编译过程，同时执行时也不像用户函数那样每一个指令都调用一次C语言编写的handler函数，因此，内部函数的执行效率更高。除了性能上的优势，内部函数还可以拥有更高的控制权限，可发挥的作用也更大，能够完成很多用户函数无法实现的功能。

前面介绍PHP函数的编译时曾经详细介绍过PHP函数的实现，函数通过 `zend_function` 来表示，这是一个联合体，用户函数使用 `zend_function.op_array` ，内部函数使用 `zend_function.internal_function` ，两者具有相同的头部用来记录函数的基本信息。不管是用户函数还是内部函数，其最终都被注册到EG(function_table)中，函数被调用时根据函数名称向这个符号表中查找。从内部函数的注册、使用过程可以看出，其定义实际非常简单，我们只需要定义一个 `zend_internal_function` 结构，然后注册到EG(function_table)中即可，接下来再重新看下内部函数的结构：

```
typedef struct _zend_internal_function {
    /* Common elements */
    zend_uchar type;
    zend_uchar arg_flags[3]; /* bitset of arg_info.pass_by_refer
ence */
    uint32_t fn_flags;
    zend_string* function_name;
    zend_class_entry *scope;
    zend_function *prototype;
    uint32_t num_args;
    uint32_t required_num_args;
    zend_internal_arg_info *arg_info;
    /* END of common elements */

    void (*handler)(INTERNAL_FUNCTION_PARAMETERS); //函数指针，展
开：void (*handler)(zend_execute_data *execute_data, zval *return
_value)
    struct _zend_module_entry *module;
    void *reserved[ZEND_MAX_RESERVED_RESOURCES];
} zend_internal_function;
```

Common elements就是与用户函数相同的头部，用来记录函数的基本信息：函数类型、参数信息、函数名等，handler是此内部函数的具体实现，PHP提供了一个宏用于此handler的定义： `PHP_FUNCTION(function_name)` 或 `ZEND_FUNCTION()` ，展开后：

```
void *zif_function_name(zend_execute_data *execute_data, zval *r
eturn_value)
{
    ...
}
```

PHP为函数名加了"zif_"前缀，gdb调试时记得加上这个前缀；另外内部函数定义了两个参数：execute_data、return_value，execute_data不用再说了，return_value是函数的返回值，这两个值在扩展中会经常用到。

比如要在扩展中定义两个函数：my_func_1()、my_func_2()，首先是编写函数：

```
PHP_FUNCTION(my_func_1)
{
    printf("Hello, I'm my_func_1\n");
}

PHP_FUNCTION(my_func_2)
{
    printf("Hello, I'm my_func_2\n");
}
```

函数定义完了就需要向PHP注册了，这里并不需要扩展自己注册，PHP提供了一个内部函数注册结构：zend_function_entry，扩展只需要为每个内部函数生成这样一个结构，然后把它们保存到扩展 `zend_module_entry.functions` 即可，在加载扩展中会自动向EG(function_table)注册。

```
typedef struct _zend_function_entry {
    const char *fname;  //函数名称
    void (*handler)(INTERNAL_FUNCTION_PARAMETERS); //handler实现
    const struct _zend_internal_arg_info *arg_info;//参数信息
    uint32_t num_args; //参数数目
    uint32_t flags;
} zend_function_entry;
```

zend_function_entry结构可以通过 `PHP_FE()` 或 `ZEND_FE()` 定义：

```
const zend_function_entry mytest_functions[] = {
    PHP_FE(my_func_1,   NULL)
    PHP_FE(my_func_2,   NULL)
    PHP_FE_END //末尾必须加这个
};
```

这几个宏的定义为：

```
#define ZEND_FE(name, arg_info)                    ZEND_FENTRY(
name, ZEND_FN(name), arg_info, 0)
#define ZEND_FENTRY(zend_name, name, arg_info, flags)   { #zend_
name, name, arg_info, (uint32_t) (sizeof(arg_info)/sizeof(struct
 _zend_internal_arg_info)-1), flags },
#define ZEND_FN(name) zif_##name
```

最后将 `zend_module_entry->functions` 设置为 `mytest_functions` 即可：

```
zend_module_entry mytest_module_entry = {
    STANDARD_MODULE_HEADER,
    "mytest",
    mytest_functions, //functions
    NULL, //PHP_MINIT(mytest),
    NULL, //PHP_MSHUTDOWN(mytest),
    NULL, //PHP_RINIT(mytest),
    NULL, //PHP_RSHUTDOWN(mytest),
    NULL, //PHP_MINFO(mytest),
    "1.0.0",
    STANDARD_MODULE_PROPERTIES
};
```

下面来测试下这两个函数能否使用，编译安装后在PHP脚本中调用这两个函数：

```
//test.php
my_func_1();
my_func_2();
```

cli模式下执行 `php test.php` 将输出：

```
Hello, I'm my_func_1
Hello, I'm my_func_2
```

大功告成，函数已经能够正常工作了，后续的工作就是不断完善handler实现扩展自己的功能了。

7.6.2 函数参数解析

上面我们定义的函数没有接收任何参数，那么扩展定义的内部函数如何读取参数呢？首先回顾下函数参数的实现：用户自定义函数在编译时会为每个参数创建一个 `zend_arg_info` 结构，这个结构用来记录参数的名称、是否引用传参、是否为可变参数等，在存储上函数参数与局部变量相同，都分配在zend_execute_data上，且最先分配的就是函数参数，调用函数时首先会进行参数传递，按参数次序依次将参数的value从调用空间传递到被调函数的zend_execute_data，函数内部像访问普通局部变量一样通过存储位置访问参数，这是用户自定义函数的参数实现。

内部函数与用户自定义函数最大的不同在于内部函数就是一个普通的C函数，除函数参数以外在zend_execute_data上没有其他变量的分配，函数参数是从PHP用户空间传到函数的，它们与用户自定义函数完全相同，包括参数的分配方式、传参过程，也是按照参数次序依次分配在zend_execute_data上，所以在扩展中定义的函数直接按照顺序从zend_execute_data上读取对应的值即可，PHP中通过 `zend_parse_parameters()` 这个函数解析zend_execute_data上保存的参数：

```
zend_parse_parameters(int num_args, const char *type_spec, ...);
```

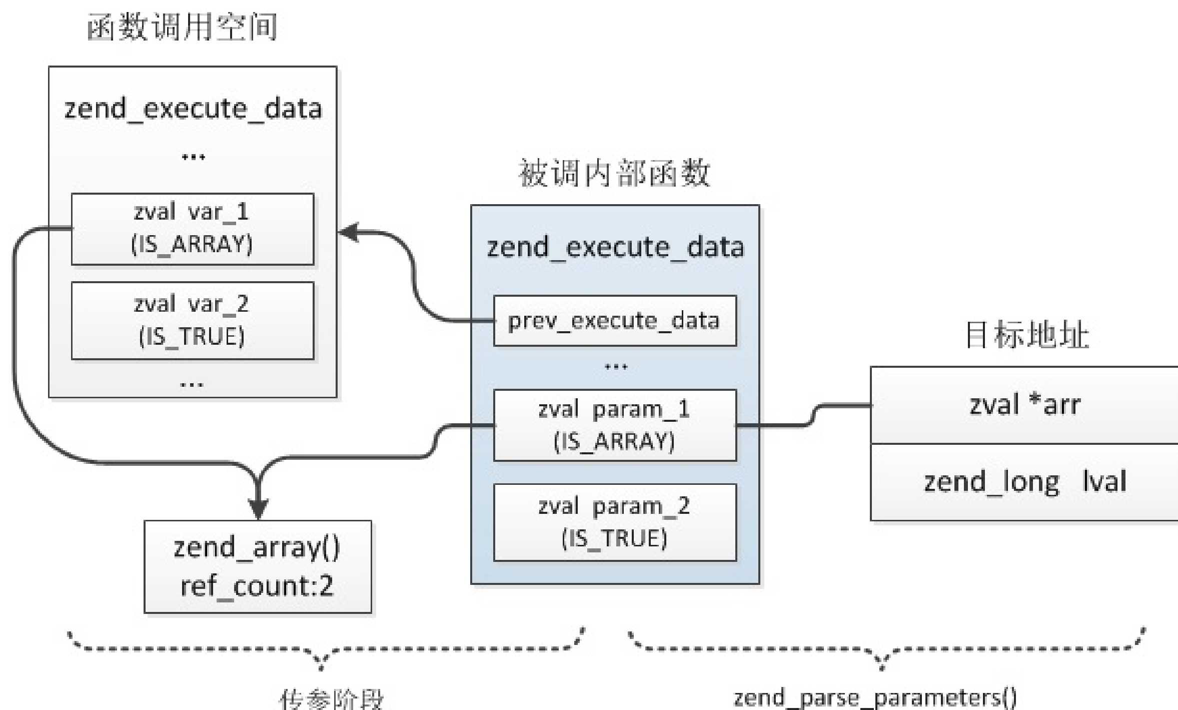
- num_args为实际传参数，通过 `ZEND_NUM_ARGS()` 获取：zend_execute_data->This.u2.num_args，前面曾介绍过 `zend_execute_data->This` 这个zval的用途；
- type_spec是一个字符串，用来标识解析参数的类型，比如:"la"表示第一个参数为整形，第二个为数组，将按照这个解析到指定变量；
- 后面是一个可变参数，用来指定解析到的变量，这个值与type_spec配合使用，即type_spec用来指定解析的变量类型，可变参数用来指定要解析到的变量，这个值必须是指针。

i解析的过程也比较容易理解，调用函数时首先会把参数拷贝到调用函数的zend_execute_data上，所以解析的过程就是按照type_spec指定的各个类型，依次从zend_execute_data上获取参数，然后将参数地址赋给目标变量，比如下面这个例子：

```
PHP_FUNCTION(my_func_1)
{
    zend_long   lval;
    zval        *arr;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "la", &lval, &arr)
 == FAILURE){
        RETURN_FALSE;
    }
    ...
}
```

对应的内存关系：



注意：解析时除了整形、浮点型、布尔型是直接硬拷贝value外，其它解析到的变量只能是指针，arr为zend_execute_data上param_1的地址，即： `zval *arr = &param_1` ，也就是说参数始终存储在zend_execute_data上，解析获取的是这些参数的地址。 `zend_parse_parameters()` 调用了 `zend_parse_va_args()` 进行处理，简单看下解析过程：

```
//va就是定义的要解析到的各个变量的地址
static int zend_parse_va_args(int num_args, const char *type_spe
c, va_list *va, int flags)
```

```
{
    const  char *spec_walk;
    int min_num_args = -1; //最少参数数
    int max_num_args = 0; //要解析的参数总数
    int post_varargs = 0;
    zval *arg;
    int arg_count; //实际传参数

    //遍历type_spec计算出min_num_args、max_num_args
    for (spec_walk = type_spec; *spec_walk; spec_walk++) {
        ...
    }
    ...
    //检查数目是否合法
    if (num_args < min_num_args || (num_args > max_num_args && m
ax_num_args >= 0)) {
        ...
    }
    //获取实际传参数：zend_execute_data.This.u2.num_args
    arg_count = ZEND_CALL_NUM_ARGS(EG(current_execute_data));
    ...
    i = 0;
    //逐个解析参数
    while (num_args-- > 0) {
        ...
        //获取第i个参数的zval地址：arg就是在zend_execute_data上分配的
局部变量
        arg = ZEND_CALL_ARG(EG(current_execute_data), i + 1);

        //解析第i个参数
        if (zend_parse_arg(i+1, arg, va, &type_spec, flags) == F
AILURE) {
            if (varargs && *varargs) {
                *varargs = NULL;
            }
            return FAILURE;
        }
        i++;
    }
}
```

接下来详细看下不同类型的解析方式。

7.6.2.1 整形：l、L

整形通过"l"、"L"标识，表示解析的参数为整形，解析到的变量类型必须是 `zend_long` ，不能解析其它类型，如果输入的参数不是整形将按照类型转换规则将其转为整形：

```
zend_long   lval;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "l", &lval){
    ...
}
printf("lval:%d\n", lval);
```

如果在标识符后加"!"，即："l!"、"L!"，则必须再提供一个zend_bool变量的地址，通过这个值可以判断传入的参数是否为NULL，如果为NULL则将要解析到的zend_long值设置为0，同时zend_bool设置为1：

```
zend_long   lval; //如果参数为NULL则此值被设为0
zend_bool   is_null; //如果参数为NULL则此值为1，否则为0

if(zend_parse_parameters(ZEND_NUM_ARGS(), "l!", &lval, &is_null)
{
    ...
}
```

具体的解析过程：

```
//zend_API.c #line:519
case 'l':
case 'L':
{
    //这里获取解析到的变量地址取的是zend_long *，所以只能解析到zend_long
    zend_long *p = va_arg(*va, zend_long *);
    zend_bool *is_null = NULL;

    //后面加"!"时check_null为1
    if (check_null) {
        is_null = va_arg(*va, zend_bool *);
    }

    if (!zend_parse_arg_long(arg, p, is_null, check_null, c == '
L')) {
        return "integer";
    }
}
```

```
static zend_always_inline int zend_parse_arg_long(zval *arg, zen
d_long *dest, zend_bool *is_null, int check_null, int cap)
{
    if (check_null) {
        *is_null = 0;
    }
    if (EXPECTED(Z_TYPE_P(arg) == IS_LONG)) {
        //传参为整形，无需转化
        *dest = Z_LVAL_P(arg);
    } else if (check_null && Z_TYPE_P(arg) == IS_NULL) {
        //传参为NULL
        *is_null = 1;
        *dest = 0;
    } else if (cap) {
        //"L"的情况
        return zend_parse_arg_long_cap_slow(arg, dest);
    } else {
        //"l"的情况
        return zend_parse_arg_long_slow(arg, dest);
    }
    return 1;
}
```

> **Note:** "l"与"L"的区别在于，当传参不是整形且转为整形后超过了整形的大小范围时，"L"将值调整为整形的最大或最小值，而"l"将报错，比如传的参数是字符串"9223372036854775808"(0x7FFFFFFFFFFFFFFF + 1)，转整形后超过了有符号int64的最大值：0x7FFFFFFFFFFFFFFF，所以如果是"L"将解析为0x7FFFFFFFFFFFFFFF。

7.6.2.2 布尔型：b

通过"b"标识符表示将传入的参数解析为布尔型，解析到的变量必须是zend_bool：

```
zend_bool   ok;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "b", &ok, &is_null) ==
 FAILURE){
    ...
}
```

"b!"的用法与整形的完全相同，也必须再提供一个zend_bool的地址用于获取传参是否为NULL，如果为NULL，则zend_bool为0，用于获取是否NULL的zend_bool为1。

7.6.2.3 浮点型：d

通过"d"标识符表示将参数解析为浮点型，解析的变量类型必须为double：

```
double  dval;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "d", &dval) == FAILURE
){
    ...
}
```

具体解析过程不再展开，"d!"与整形、布尔型用法完全相同。

7.6.2.4 字符串：s、S、p、P

字符串解析有两种形式：char、*zend_string*，其中*"s"*将参数解析到`*char*`，且需要额外提供一个size_t类型的变量用于获取字符串长度，"S"将解析到zend_string：

```
char    *str;
size_t  str_len;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "s", &str, &str_len) =
= FAILURE){
    ...
}
```

```
zend_string    *str;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "S", &str) == FAILURE)
{
    ...
}
```

"s!"、"S!"与整形、布尔型用法不同，字符串时不需要额外提供zend_bool的地址，如果参数为NULL，则char*、zend_string将设置为NULL。除了"s"、"S"之外还有两个类似的："p"、"P"，从解析规则来看主要用于解析路径，实际与普通字符串没什么区别，尚不清楚这俩有什么特殊用法。

7.6.2.5 数组：a、A、h、H

数组的解析也有两类，一类是解析到zval层面，另一类是解析到HashTable，其中"a"、"A"解析到的变量必须是zval，"h"、"H"解析到HashTable，这两类是等价的：

```
zval        *arr;   //必须是zval指针，不能是zval arr，因为参数保存在zend_execute_data上，arr为此空间上参数的地址
HashTable   *ht;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "ah", &arr, &ht) == FAILURE){
    ...
}
```

具体解析过程：

```
case 'A':
case 'a':
{
    //解析到zval *
    zval **p = va_arg(*va, zval **);

    if (!zend_parse_arg_array(arg, p, check_null, c == 'A')) {
        return "array";
    }
}
break;

case 'H':
case 'h':
{
    //解析到HashTable *
    HashTable **p = va_arg(*va, HashTable **);

    if (!zend_parse_arg_array_ht(arg, p, check_null, c == 'H'))
{
        return "array";
    }
}
break;
```

"a!"、"A!"、"h!"、"H!"的用法与字符串一致，也不需要额外提供别的地址，如果传参为NULL，则对应解析到的zval、*HashTable*也为NULL。

> **Note:**
>
> 1、"a"与"A"当传参为数组时没有任何差别，它们的区别在于：如果传参为对象"A"将按照对象解析到zval，而"a"将报错
>
> 2、"h"与"H"当传参为数组时同样没有差别，当传参为对象时，"H"将把对象的成员参数数组解析到目标变量，"h"将报错

7.6.2.6 对象：o、O

如果参数是一个对象则可以通过"o"、"O"将其解析到目标变量，注意：只能解析为zval，无法解析为*zend_object*。

```
zval    *obj;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE)
{
    ...
}
```

"O"是要求解析指定类或其子类的对象，类似传参时显式的声明了参数类型的用法：`function my_func(MyClass $obj){...}`，如果参数不是指定类的实例化对象则无法解析。

"o!"、"O!"与字符串用法相同。

7.6.2.7 资源：r

如果参数为资源则可以通过"r"获取其zval的地址，但是无法直接解析到zend_resource的地址，与对象相同。

```
zval    *res;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "r", &res) == FAILURE)
{
    ...
}
```

"r!"与字符串用法相同。

7.6.2.8 类：C

如果参数是一个类则可以通过"C"解析出zend_class_entry地址：`function my_func(stdClass){...}`，这里有个地方比较特殊，解析到的变量可以设定为一个类，这种情况下解析时将会找到的类与指定的类之间的父子关系，只有存在父子关系才能解析，如果只是想根据参数获取类型的zend_class_entry地址，记得将解析到的地址初始化为NULL，否则将会不可预料的错误。

```
zend_class_entry    *ce = NULL; //初始为NULL

if(zend_parse_parameters(ZEND_NUM_ARGS(), "C", &ce) == FAILURE){
    RETURN_FALSE;
}
```

7.6.2.9 callable：f

callable指函数或成员方法，如果参数是函数名称字符串、array(对象/类,成员方法)，则可以通过"f"标识符解析出 `zend_fcall_info` 结构，这个结构是调用函数、成员方法时的唯一输入。

```
zend_fcall_info         callable; //注意，这两个结构不能是指针
zend_fcall_info_cache   call_cache;

if(zend_parse_parameters(ZEND_NUM_ARGS(), "f", &callable, &call_
cache) == FAILURE){
    RETURN_FALSE;
}
```

函数调用：

```
my_func_1("func_name");
//或
my_func_1(array('class_name', 'static_method'));
//或
my_func_1(array($object, 'method'));
```

解析出 `zend_fcall_info` 后就可以通过 `zend_call_function()` 调用函数、成员方法了，提供"f"解析到 `zend_fcall_info` 的用意是简化函数调用的操作，否则需要我们自己去查找函数、检查是否可被调用等工作，关于这个结构稍后介绍函数调用时再作详细说明。

7.6.2.10 任意类型：z

"z"表示按参数实际类型解析，比如参数为字符串就解析为字符串，参数为数组就解析为数组，这种实际就是将zend_execute_data上的参数地址拷贝到目的变量了，没有做任何转化。

"z!"与字符串用法相同。

7.6.2.11 其它标识符

除了上面介绍的这些解析符号以外，还有几个有特殊用法的标识符："|"、"+"、"*"，它们并不是用来表示某种数据类型的。

- **|**： 表示此后的参数为可选参数，可以不传，比如解析规则为："al|b"，则可以传2个或3个参数，如果是："alb"，则必须传3个，否则将报错
- **+**、*****： 用于可变参数， `+、*` 的区别在于 * 表示可以不传可变参数，而 + 表示可变参数至少有一个。可变参数将被解析到zval数组，可以通过一个整形参数，用于获取具体的数量，例如：

```
PHP_FUNCTION(my_func_1)
{
  zval *args;
  int argc;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "+", &args, &argc) == FAILURE) {
      return;
  }
  //...
}
```

argc获取的就是可变参数的数量，args为参数数组，指向第一个参数，可以通过args[i]获取其它参数，比如这样传参：

```
my_func_1(array(), 1, false, "ddd");
```

那么传入的4个参数就可以在解析后通过args[0]、args[1]、args[2]、args[3]获取。

7.6.3 引用传参

上一节介绍了如何在内部函数中解析参数，这里还有一种情况没有讲到，那就是引用传参：

```
$a = array();

function my_func(&$a){
    $a[] = 1;
}
```

上面这个例子在函数中对$a的修改将反映到原变量上，那么这种用法如何在内部函数中实现呢？上一节介绍参数解析的过程中并没有提到用户函数中参数的zend_arg_info结构，内部函数中也有类似的一个结构用于函数注册时指定参数的一些信息：zend_internal_arg_info。

```
typedef struct _zend_internal_arg_info {
    const char *name;               //参数名
    const char *class_name;
    zend_uchar type_hint;           //显式声明的类型
    zend_uchar pass_by_reference;   //是否引用传参
    zend_bool allow_null;           //是否允许参数为NULL，类似"!"的
用法
    zend_bool is_variadic;          //是否为可变参数
} zend_internal_arg_info;
```

这个结构几乎与zend_arg_info完全一样，不同的地方只在于name、class_name的类型，zend_arg_info这两个成员的类型都是zend_string。如果函数需要使用引用类型的参数或返回引用就需要创建函数的参数数组，这个数组通过：`ZEND_BEGIN_ARG_INFO()`或`ZEND_BEGIN_ARG_INFO_EX()`、`ZEND_END_ARG_INFO()`宏定义：

```
#define ZEND_BEGIN_ARG_INFO_EX(name, _unused, return_reference,
required_num_args)
#define ZEND_BEGIN_ARG_INFO(name, _unused)
```

- **name:** 参数数组名，注册函数`PHP_FE(function, arg_info)`会用到

- **_unused:** 保留值，暂时无用
- **return_reference:** 返回值是否为引用，一般很少会用到
- **required_num_args:** required参数数

这两个宏需要与`ZEND_END_ARG_INFO()`配合使用：

```
ZEND_BEGIN_ARG_INFO_EX(arginfo_my_func_1, 0, 0, 2)
    ...
ZEND_END_ARG_INFO()
```

接着就是在上面两个宏中间定义每一个参数的zend_internal_arg_info，PHP提供的宏有：

```
//pass_by_ref表示是否引用传参，name为参数名称
#define ZEND_ARG_INFO(pass_by_ref, name)
     { #name, NULL, 0, pass_by_ref, 0, 0 },

//只声明此参数为引用传参
#define ZEND_ARG_PASS_INFO(pass_by_ref)
     { NULL,  NULL, 0, pass_by_ref, 0, 0 },

//显式声明此参数的类型为指定类的对象，等价于PHP中这样声明：MyClass $obj
#define ZEND_ARG_OBJ_INFO(pass_by_ref, name, classname, allow_nu
ll)  { #name, #classname, IS_OBJECT, pass_by_ref, allow_null, 0
},

//显式声明此参数类型为数组，等价于：array $arr
#define ZEND_ARG_ARRAY_INFO(pass_by_ref, name, allow_null)
     { #name, NULL, IS_ARRAY, pass_by_ref, allow_null, 0 },

//显式声明为callable，将检查函数、成员方法是否可调
#define ZEND_ARG_CALLABLE_INFO(pass_by_ref, name, allow_null)
     { #name, NULL, IS_CALLABLE, pass_by_ref, allow_null, 0 },

//通用宏，自定义各个字段
#define ZEND_ARG_TYPE_INFO(pass_by_ref, name, type_hint, allow_n
ull) { #name, NULL, type_hint, pass_by_ref, allow_null, 0 },

//声明为可变参数
#define ZEND_ARG_VARIADIC_INFO(pass_by_ref, name)
     { #name, NULL, 0, pass_by_ref, 0, 1 },
```

举个例子来看：

```
function my_func_1(&$a, Exception $c){
    ...
}
```

用内核实现则可以这么定义：

```
ZEND_BEGIN_ARG_INFO_EX(arginfo_my_func_1, 0, 0, 1)
    ZEND_ARG_INFO(1, a) //引用
    ZEND_ARG_OBJ_INFO(0, b, Exception, 0) //注意：这里不要把字符串加
""
ZEND_END_ARG_INFO()
```

展开后：

```
static const zend_internal_arg_info name[] = {
    //多出来的这个是给返回值用的
    { (const char*)(zend_uintptr_t)(2), NULL, 0, 0, 0, 0 },
    { "a", NULL, 0, 0, 0, 0 },
    { "b", "Exception", 8, 1, 0, 0 },
}
```

第一个数组元素用于记录必传参数的数量以及返回值是否为引用。定义完这个数组接下来就需要把这个数组告诉函数：

```
const zend_function_entry mytest_functions[] = {
    PHP_FE(my_func_1,   arginfo_my_func_1)
    PHP_FE(my_func_2,   NULL)
    PHP_FE_END //末尾必须加这个
};
```

引用参数通过 `zend_parse_parameters()` 解析时只能使用"z"解析，不能再直接解析为zend_value了，否则引用将失效：

```
PHP_FUNCTION(my_func_1)
{
    zval    *lval; //必须为zval，定义为zend_long也能解析出，但不是引用
    zval    *obj;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "zo", &lval, &obj)
 == FAILURE){
        RETURN_FALSE;
    }

    //lval的类型为IS_REFERENCE
    zval *real_val = Z_REFVAL_P(lval); //获取实际引用的zval地址：&(
lval.value->ref.val)
    Z_LVAL_P(real_val) = 100; //设置实际引用的类型
}
```

```
$a = 90;
$b = new Exception;
my_func_1($a, $b);

echo $a;
==========[output]===========
100
```

> **Note:** 参数数组与zend_parse_parameters()有很多功能重合，两者都会生效，对zend_internal_arg_info验证在zend_parse_parameters()之前，为避免混乱两者应该保持一致；另外，虽然内部函数的参数数组并不强制定义声明，但还是建议声明。

7.6.4 函数返回值

调用内部函数时其返回值指针作为参数传入，这个参数为 `zval *return_value` ，如果函数有返回值直接设置此指针即可，需要特别注意的是设置返回值时需要增加其引用计数，举个例子来看：

```
PHP_FUNCTION(my_func_1)
{
    zval    *arr;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "a", &arr) == FAIL
URE){
        RETURN_FALSE;
    }

    //增加引用计数
    Z_ADDREF_P(arr);

    //设置返回值为数组：
    ZVAL_ARR(return_value, Z_ARR_P(arr));
}
```

此函数接收一个数组，然后直接返回该数组，相当于：

```
function my_func_1($arr){
    return $arr;
}
```

调用该函数：

```
$a = array();        //$a -> zend_array(refcount:1)
$b = my_func_1($a);  //传参后：参数arr -> zend_array(refcount:2)
                     //然后函数内部赋给了返回值:$b,$a,arr -> zend_arr
ay(refcount:3)
                     //函数return阶段释放了参数：$b,$a -> zend_array
(refcount:2)
var_dump($b);
=============[output]===========
array(0) {
}
```

虽然可以直接设置return_value，但实际使用时并不建议这么做，因为PHP提供了很多专门用于设置返回值的宏，这些宏定义在 `zend_API.h` 中：

```
//返回布尔型，b：IS_FALSE、IS_TRUE
#define RETURN_BOOL(b)                  { RETVAL_BOOL(b); return; }

//返回NULL
#define RETURN_NULL()                   { RETVAL_NULL(); return;}


//返回整形，l类型：zend_long
#define RETURN_LONG(l)                  { RETVAL_LONG(l); return; }

//返回浮点值，d类型：double
#define RETURN_DOUBLE(d)                { RETVAL_DOUBLE(d); return; }

//返回字符串，可返回内部字符串，s类型为：zend_string *
#define RETURN_STR(s)                   { RETVAL_STR(s); return; }

//返回内部字符串，这种变量将不会被回收，s类型为：zend_string *
#define RETURN_INTERNED_STR(s)          { RETVAL_INTERNED_STR(s); return; }

//返回普通字符串，非内部字符串，s类型为：zend_string *
#define RETURN_NEW_STR(s)               { RETVAL_NEW_STR(s); return; }

//拷贝字符串用于返回，这个会自己加引用计数，s类型为：zend_string *
#define RETURN_STR_COPY(s)              { RETVAL_STR_COPY(s); return; }

//返回char *类型的字符串，s类型为char *
#define RETURN_STRING(s)                { RETVAL_STRING(s); return; }

//返回char *类型的字符串，s类型为char *，l为字符串长度，类型为size_t
#define RETURN_STRINGL(s, l)            { RETVAL_STRINGL(s, l); return; }
```

```
//返回空字符串
#define RETURN_EMPTY_STRING()          { RETVAL_EMPTY_STRING();
 return; }

//返回资源，r类型：zend_resource *
#define RETURN_RES(r)                  { RETVAL_RES(r); return;
 }

//返回数组，r类型：zend_array *
#define RETURN_ARR(r)                  { RETVAL_ARR(r); return;
 }

//返回对象，r类型：zend_object *
#define RETURN_OBJ(r)                  { RETVAL_OBJ(r); return;
 }

//返回zval
#define RETURN_ZVAL(zv, copy, dtor)    { RETVAL_ZVAL(zv, copy,
dtor); return; }

//返回false
#define RETURN_FALSE                   { RETVAL_FALSE; return; }


//返回true
#define RETURN_TRUE                    { RETVAL_TRUE; return; }
```

7.6.5 函数调用

实际应用中，扩展可能需要调用用户自定义的函数或者其他扩展定义的内部函数，前面章节已经介绍过函数的执行过程，这里不再重复，本节只介绍下PHP提供的函数调用API的使用:

```
ZEND_API int call_user_function(HashTable *function_table, zval
*object, zval *function_name, zval *retval_ptr, uint32_t param_c
ount, zval params[]);
```

各参数的含义：

- **function_table:** 函数符号表，普通函数是EG(function_table)，如果是成员方法则是zend_class_entry.function_table
- **object:** 调用成员方法时的对象
- **function_name:** 调用的函数名称
- **retval_ptr:** 函数返回值地址
- **param_count:** 参数数量
- **params:** 参数数组

从接口的定义看其使用还是很简单的，不需要我们关心执行过程中各阶段复杂的操作。下面从一个具体的例子看下其使用：

（1）在PHP中定义了一个普通的函数，将参数$i加上100后返回：

```
function mySum($i){
    return $i+100;
}
```

（2）接下来在扩展中调用这个函数：

```
PHP_FUNCTION(my_func_1)
{
    zend_long   i;
    zval        call_func_name, call_func_ret, call_func_params[1
];
    uint32_t    call_func_param_cnt = 1;
    zend_string *call_func_str;
    char        *func_name = "mySum";

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "l", &i) == FAILUR
E){
        RETURN_FALSE;
    }

    //分配zend_string:调用完需要释放
    call_func_str = zend_string_init(func_name, strlen(func_name
), 0);
    //设置到zval
    ZVAL_STR(&call_func_name, call_func_str);

    //设置参数
    ZVAL_LONG(&call_func_params[0], i);

    //call
    if(SUCCESS != call_user_function(EG(function_table), NULL, &
call_func_name, &call_func_ret, call_func_param_cnt, call_func_p
arams)){
        zend_string_release(call_func_str);
        RETURN_FALSE;
    }
    zend_string_release(call_func_str);
    RETURN_LONG(Z_LVAL(call_func_ret));
}
```

（3）最后调用这个内部函数：

```
function mySum($i){
    return $i+100;
}

echo my_func_1(60);
===========[output]===========
160
```

`call_user_function()` 并不是只能调用PHP脚本中定义的函数，内核或其它扩展注册的函数同样可以通过此函数调用，比如：array_merge()。

```
PHP_FUNCTION(my_func_1)
{
    zend_array  *arr1, *arr2;
    zval        call_func_name, call_func_ret, call_func_params[2
];
    uint32_t    call_func_param_cnt = 2;
    zend_string *call_func_str;
    char        *func_name = "array_merge";

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "hh", &arr1, &arr2
) == FAILURE){
        RETURN_FALSE;
    }
    //分配zend_string
    call_func_str = zend_string_init(func_name, strlen(func_name
), 0);
    //设置到zval
    ZVAL_STR(&call_func_name, call_func_str);

    ZVAL_ARR(&call_func_params[0], arr1);
    ZVAL_ARR(&call_func_params[1], arr2);

    if(SUCCESS != call_user_function(EG(function_table), NULL, &
call_func_name, &call_func_ret, call_func_param_cnt, call_func_p
arams)){
        zend_string_release(call_func_str);
        RETURN_FALSE;
    }
    zend_string_release(call_func_str);
    RETURN_ARR(Z_ARRVAL(call_func_ret));
}
```

```
$arr1 = array(1,2);
$arr2 = array(3,4);

$arr = my_func_1($arr1, $arr2);
var_dump($arr);
```

你可能会注意到，上面的例子通过 `call_user_function()` 调用函数时并没有增加两个数组参数的引用计数，但根据前面介绍的内容：函数传参时不会硬拷贝value，而是增加参数value的引用计数，然后在函数return阶段再把引用减掉。实际是 `call_user_function()` 替我们完成了这个工作，下面简单看下其处理过程。

```
int call_user_function(HashTable *function_table, zval *object,
zval *function_name, zval *retval_ptr, uint32_t param_count, zva
l params[])
{
    return call_user_function_ex(function_table, object, functio
n_name, retval_ptr, param_count, params, 1, NULL);
}

int call_user_function_ex(HashTable *function_table, zval *objec
t, zval *function_name, zval *retval_ptr, uint32_t param_count,
zval params[], int no_separation, zend_array *symbol_table)
{
    zend_fcall_info fci;

    fci.size = sizeof(fci);
    fci.function_table = function_table;
    fci.object = object ? Z_OBJ_P(object) : NULL;
    ZVAL_COPY_VALUE(&fci.function_name, function_name);
    fci.retval = retval_ptr;
    fci.param_count = param_count;
    fci.params = params;
    fci.no_separation = (zend_bool) no_separation;
    fci.symbol_table = symbol_table;

    return zend_call_function(&fci, NULL);
}
```

`call_user_function()` 将我们提供的参数组装为 `zend_fcall_info` 结构，然后调用 `zend_call_function()` 进行处理，还记得 `zend_parse_parameters()` 那个"f"解析符吗？它也是将输入的函数名称解析为一个 `zend_fcall_info` ，可以更方便的调用函数，同时我们也可以自己创建一个 `zend_fcall_info` 结构，然后使用 `zend_call_function()` 完成函数的调用。

```
int zend_call_function(zend_fcall_info *fci, zend_fcall_info_cac
he *fci_cache)
{
    ...
    for (i=0; i<fci->param_count; i++) {
        zval *param;
        zval *arg = &fci->params[i];
        ...
        //为参数添加引用
        if (Z_OPT_REFCOUNTED_P(arg)) {
            Z_ADDREF_P(arg);
        }
    }
    ...
    //调用的是用户函数
    if (func->type == ZEND_USER_FUNCTION) {
        //执行
        zend_init_execute_data(call, &func->op_array, fci->retva
l);
        zend_execute_ex(call);
    }else if (func->type == ZEND_INTERNAL_FUNCTION){ //内部函数
        if (EXPECTED(zend_execute_internal == NULL)) {
            func->internal_function.handler(call, fci->retval);
        } else {
            zend_execute_internal(call, fci->retval);
        }
    }
    ...
}
```

7.7 zval的操作

扩展中经常会用到各种类型的zval，PHP提供了很多宏用于不同类型zval的操作，尽管我们也可以自己操作zval，但这并不是一个好习惯，因为zval有很多其它用途的标识，如果自己去管理这些值将是非常繁琐的一件事，所以我们应该使用PHP提供的这些宏来操作用到的zval。

7.7.1 新生成各类型zval

PHP7将变量的引用计数转移到了具体的value上，所以zval更多的是作为统一的传输格式，很多情况下只是临时性使用，比如函数调用时的传参，最终需要的数据是zval携带的zend_value，函数从zval取得zend_value后就不再关心zval了，这种就可以直接在栈上分配zval。分配完zval后需要将其设置为我们需要的类型以及设置其zend_value，PHP中定义的 `ZVAL_XXX()` 系列宏就是用来干这个的，这些宏第一个参数z均为要设置的zval的指针，后面为要设置的zend_value。

- **ZVAL_UNDEF(z):** 表示zval被销毁
- **ZVAL_NULL(z):** 设置为NULL
- **ZVAL_FALSE(z):** 设置为false
- **ZVAL_TRUE(z):** 设置为true
- **ZVAL_BOOL(z, b):** 设置为布尔型，b为IS_TRUE、IS_FALSE，与上面两个等价
- **ZVAL_LONG(z, l):** 设置为整形，l类型为zend_long，如：`zval z; ZVAL_LONG(&z, 88);`
- **ZVAL_DOUBLE(z, d):** 设置为浮点型，d类型为double
- **ZVAL_STR(z, s):** 设置字符串，将z的value设置为s，s类型为zend_string*，不会增加s的refcount，支持interned strings
- **ZVAL_NEW_STR(z, s):** 同ZVAL_STR(z, s)，s为普通字符串，不支持interned strings
- **ZVAL_STR_COPY(z, s):** 将s拷贝到z的value，s类型为zend_string*，同ZVAL_STR(z, s)，这里会增加s的refcount
- **ZVAL_ARR(z, a):** 设置为数组，a类型为zend_array*
- **ZVAL_NEW_ARR(z):** 新分配一个数组，主动分配一个zend_array
- **ZVAL_NEW_PERSISTENT_ARR(z):** 创建持久化数组，通过malloc分配，需要手动释放

- **ZVAL_OBJ(z, o):** 设置为对象，o类型为zend_object*
- **ZVAL_RES(z, r):** 设置为资源，r类型为zend_resource*
- **ZVAL_NEW_RES(z, h, p, t):** 新创建一个资源，h为资源handle，t为type，p为资源ptr指向结构
- **ZVAL_REF(z, r):** 设置为引用，r类型为zend_reference*
- **ZVAL_NEW_EMPTY_REF(z):** 新创建一个空引用，没有设置具体引用的value
- **ZVAL_NEW_REF(z, r):** 新创建一个引用，r为引用的值，类型为zval*
- ...

7.7.2 获取zval的值及类型

zval的类型通过 `Z_TYPE(zval)` 、 `Z_TYPE_P(zval*)` 两个宏获取，这个值取的就是 `zval.u1.v.type` ，但是设置时不要只修改这个type，而是要设置typeinfo，因为zval还有其它的标识需要设置，比如是否使用引用计数、是否可被垃圾回收、是否可被复制等等。

内核提供了 `Z_XXX(zval)` 、 `Z_XXX_P(zval*)` 系列的宏用于获取不同类型zval的value。

- **Z_LVAL(zval)**、**Z_LVAL_P(zval_p):** 返回zend_long
- **Z_DVAL(zval)**、**Z_DVAL_P(zval_p):** 返回double
- **Z_STR(zval)**、**Z_STR_P(zval_p):** 返回zend_string*
- **Z_STRVAL(zval)**、**Z_STRVAL_P(zval_p):** 返回char*，即：zend_string->val
- **Z_STRLEN(zval)**、**Z_STRLEN_P(zval_p):** 获取字符串长度
- **Z_STRHASH(zval)**、**Z_STRHASH_P(zval_p):** 获取字符串的哈希值
- **Z_ARR(zval)**、**Z_ARR_P(zval_p)**、**Z_ARRVAL(zval)**、**Z_ARRVAL_P(zval_p):** 返回zend_array*
- **Z_OBJ(zval)**、**Z_OBJ_P(zval_p):** 返回zend_object*
- **Z_OBJ_HT(zval)**、**Z_OBJ_HT_P(zval_p):** 返回对象的zend_object_handlers，即zend_object->handlers
- **Z_OBJ_HANDLER(zval, hf)**、**Z_OBJ_HANDLER_P(zv_p, hf):** 获取对象各操作的handler指针，hf为write_property、read_property等，注意：这个宏取到的为只读，不要试图修改这个值(如：Z_OBJ_HANDLER(obj, write_property) = xxx;)，因为对象的handlers成员前加了const修饰符
- **Z_OBJCE(zval)**、**Z_OBJCE_P(zval_p):** 返回对象的zend_class_entry*
- **Z_OBJPROP(zval)**、**Z_OBJPROP_P(zval_p):** 获取对象的成员数组
- **Z_RES(zval)**、**Z_RES_P(zval_p):** 返回zend_resource*

- **Z_RES_HANDLE(zval)、Z_RES_HANDLE_P(zval_p):** 返回资源handle
- **Z_RES_TYPE(zval)、Z_RES_TYPE_P(zval_p):** 返回资源type
- **Z_RES_VAL(zval)、Z_RES_VAL_P(zval_p):** 返回资源ptr
- **Z_REF(zval)、Z_REF_P(zval_p):** 返回zend_reference*
- **Z_REFVAL(zval)、Z_REFVAL_P(zval_p):** 返回引用的zval*

除了这些与PHP变量类型相关的宏之外，还有一些内核自己使用类型的宏：

```
//获取indirect的zval，指向另一个zval
#define Z_INDIRECT(zval)                (zval).value.zv
#define Z_INDIRECT_P(zval_p)            Z_INDIRECT(*(zval_p))

#define Z_CE(zval)                      (zval).value.ce
#define Z_CE_P(zval_p)                  Z_CE(*(zval_p))

#define Z_FUNC(zval)                    (zval).value.func
#define Z_FUNC_P(zval_p)                Z_FUNC(*(zval_p))

#define Z_PTR(zval)                     (zval).value.ptr
#define Z_PTR_P(zval_p)                 Z_PTR(*(zval_p))
```

7.7.3 类型转换

```
//将原类型转为特定类型，会更改原来的值
ZEND_API void ZEND_FASTCALL convert_to_long(zval *op);
ZEND_API void ZEND_FASTCALL convert_to_double(zval *op);
ZEND_API void ZEND_FASTCALL convert_to_long_base(zval *op, int b
ase);
ZEND_API void ZEND_FASTCALL convert_to_null(zval *op);
ZEND_API void ZEND_FASTCALL convert_to_boolean(zval *op);
ZEND_API void ZEND_FASTCALL convert_to_array(zval *op);
ZEND_API void ZEND_FASTCALL convert_to_object(zval *op);

#define convert_to_cstring(op) if (Z_TYPE_P(op) != IS_STRING) {
_convert_to_cstring((op) ZEND_FILE_LINE_CC); }
#define convert_to_string(op) if (Z_TYPE_P(op) != IS_STRING) { _
convert_to_string((op) ZEND_FILE_LINE_CC); }

//获取格式化为long的值，不会更改原来的值，op类型为zval*，返回值为zend_long

#define zval_get_long(op) _zval_get_long((op))
//获取格式化为double的值，返回值double
#define zval_get_double(op) _zval_get_double((op))
//获取格式化为string的值，返回值zend_string *
#define zval_get_string(op) _zval_get_string((op))

//字符串转整形
ZEND_API int ZEND_FASTCALL zend_atoi(const char *str, int str_le
n);
ZEND_API zend_long ZEND_FASTCALL zend_atol(const char *str, int
str_len);

//判断是否为true
#define zval_is_true(op) \
    zend_is_true(op)
```

7.7.4 引用计数

在扩展中操作与PHP用户空间相关的变量时需要考虑是否需要对其引用计数进行加减，比如下面这个例子：

```
function test($arr){
    return $arr;
}

$a = array(1,2);
$b = test($a);
```

如果把函数test()用内部函数实现，这个函数接受了一个PHP用户空间传入的数组参数，然后又返回并赋值给了PHP用户空间的另外一个变量，这个时候就需要增加传入数组的refcount，因为这个数组由PHP用户空间分配，函数调用前refcount=1，传到内部函数时相当于赋值给了函数的参数，因此refcount增加了1变为2，这次增加在函数执行完释放参数时会减掉，等返回并赋值给$b后此时共有两个变量指向这个数组，所以内部函数需要增加refcount，增加的引用是给返回值的。test()翻译成内部函数：

```
PHP_FUNCTION(test)
{
    zval    *arr;

    if(zend_parse_parameters(ZEND_NUM_ARGS(), "a", &arr) == FAIL
URE){
        RETURN_FALSE;
    }
    //如果注释掉下面这句将导致core dumped
    Z_TRY_ADDREF_P(arr);
    RETURN_ARR(Z_ARR_P(arr));
}
```

那么在哪些情况下需要考虑设置引用计数呢？一个关键条件是：操作的是与PHP用户空间相关的变量，包括对用户空间变量的修改、赋值，要明确的一点是引用计数是用来解决多个变量指向同一个value问题的，所以在PHP中来回传递zval的时候就需要考虑下是不是要修改引用计数，下面总结下PHP中常见的会对引用计数进行操作的情况：

- **(1)**变量赋值: 变量赋值是最常见的情况，一个用到引用计数的变量类型在初始赋值时其refcount=1，如果后面把此变量又赋值给了其他变量那么就会相应的增加其引用计数

- **(2)**数组操作：如果把一个变量插入数组中那么就需要增加这个变量的引用计数，如果要删除一个数组元素则要相应的减少其引用
- **(3)**函数调用：传参实际可以当做普通的变量赋值，将调用空间的变量赋值给被调函数空间的变量，函数返回时会销毁函数空间的变量，这时又会减掉传参的引用，这两个过程由内核完成，不需要扩展自己处理
- **(4)**成员属性：当把一个变量赋值给对象的成员属性时需要增加引用计数

PHP中定义了以下宏用于引用计数的操作：

```
//获取引用数：pz类型为zval*
#define Z_REFCOUNT_P(pz)              zval_refcount_p(pz)
//设置引用数
#define Z_SET_REFCOUNT_P(pz, rc)      zval_set_refcount_p(pz, rc)
//增加引用
#define Z_ADDREF_P(pz)                zval_addref_p(pz)
//减少引用
#define Z_DELREF_P(pz)                zval_delref_p(pz)

#define Z_REFCOUNT(z)                 Z_REFCOUNT_P(&(z))
#define Z_SET_REFCOUNT(z, rc)         Z_SET_REFCOUNT_P(&(z), rc)
#define Z_ADDREF(z)                   Z_ADDREF_P(&(z))
#define Z_DELREF(z)                   Z_DELREF_P(&(z))

//只对使用了引用计数的变量类型增加引用，建议使用这个
#define Z_TRY_ADDREF_P(pz) do {       \
    if (Z_REFCOUNTED_P((pz))) {       \
        Z_ADDREF_P((pz));             \
    }                                 \
} while (0)

#define Z_TRY_DELREF_P(pz) do {       \
    if (Z_REFCOUNTED_P((pz))) {       \
        Z_DELREF_P((pz));             \
    }                                 \
} while (0)

#define Z_TRY_ADDREF(z)               Z_TRY_ADDREF_P(&(z))
#define Z_TRY_DELREF(z)               Z_TRY_DELREF_P(&(z))
```

这些宏操作类型都是zval或zval*，如果需要操作具体value的引用计数可以使用以下宏：

```
//直接获取zend_value的引用，可以直接通过这个宏修改value的refcount
#define GC_REFCOUNT(p)                (p)->gc.refcount
```

另外还有几个常用的宏：

```
//判断zval是否用到引用计数机制
#define Z_REFCOUNTED(zval)            ((Z_TYPE_FLAGS(zval) & IS_TYPE_REFCOUNTED) != 0)
#define Z_REFCOUNTED_P(zval_p)        Z_REFCOUNTED(*(zval_p))

//根据zval获取value的zend_refcounted头部
#define Z_COUNTED(zval)               (zval).value.counted
#define Z_COUNTED_P(zval_p)           Z_COUNTED(*(zval_p))
```

7.7.5 字符串操作

PHP中字符串(即：zend_string)操作相关的宏及函数：

```
//创建zend_string
zend_string *zend_string_init(const char *str, size_t len, int persistent);

//字符串复制，只增加引用
zend_string *zend_string_copy(zend_string *s);

//字符串拷贝，硬拷贝
zend_string *zend_string_dup(zend_string *s, int persistent);

//将字符串按len大小重新分配，会减少s的refcount，返回新的字符串
zend_string *zend_string_realloc(zend_string *s, size_t len, int persistent);

//延长字符串，与zend_string_realloc()类似，不同的是len不能小于s的长度
zend_string *zend_string_extend(zend_string *s, size_t len, int persistent);
```

```
//截断字符串，与zend_string_realloc()类似，不同的是len不能大于s的长度
zend_string *zend_string_truncate(zend_string *s, size_t len, int
 persistent);

//获取字符串refcount
uint32_t zend_string_refcount(const zend_string *s);

//增加字符串refcount
uint32_t zend_string_addref(zend_string *s);

//减少字符串refcount
uint32_t zend_string_delref(zend_string *s);

//释放字符串，减少refcount，为0时销毁
void zend_string_release(zend_string *s);

//销毁字符串，不管引用计数是否为0
void zend_string_free(zend_string *s);

//比较两个字符串是否相等，区分大小写，memcmp()
zend_bool zend_string_equals(zend_string *s1, zend_string *s2);

//比较两个字符串是否相等，不区分大小写
#define zend_string_equals_ci(s1, s2) \
    (ZSTR_LEN(s1) == ZSTR_LEN(s2) && !zend_binary_strcasecmp(ZST
R_VAL(s1), ZSTR_LEN(s1), ZSTR_VAL(s2), ZSTR_LEN(s2)))

//其它宏，zstr类型为zend_string*
#define ZSTR_VAL(zstr)  (zstr)->val //获取字符串
#define ZSTR_LEN(zstr)  (zstr)->len //获取字符串长度
#define ZSTR_H(zstr)    (zstr)->h   //获取字符串哈希值
#define ZSTR_HASH(zstr) zend_string_hash_val(zstr) //计算字符串哈
希值
```

除了上面这些，还有很多字符串大小转换、字符串比较的API定义在zend_operators.h中，这里不再列举。

7.7.6 数组操作

7.7.6.1 创建数组

创建一个新的HashTable分为两步：首先是分配zend_array内存，这个可以通过 `ZVAL_NEW_ARR()` 宏分配，也可以自己直接分配；然后初始化数组，通过 `zend_hash_init()` 宏完成，如果不进行初始化数组将无法使用。

```
#define zend_hash_init(ht, nSize, pHashFunction, pDestructor, pe
rsistent) \
    _zend_hash_init((ht), (nSize), (pDestructor), (persistent) Z
END_FILE_LINE_CC)
```

- **ht**： 数组地址HashTable*，如果内部使用可以直接通过emalloc分配
- **nSize**： 初始化大小，只是参考值，这个值会被对齐到2^n，最小为8
- **pHashFunction**： 无用，设置为NULL即可
- **pDestructor**： 删除或更新数组元素时会调用这个函数对操作的元素进行处理，比如将一个字符串插入数组，字符串的refcount增加，删除时不是简单的将元素的Bucket删除就可以了，还需要对其refcount进行处理，这个函数就是进行清理工作的
- **persistent**： 是否持久化

示例：

```
zval        array;
uint32_t    size;

ZVAL_NEW_ARR(&array);
zend_hash_init(Z_ARRVAL(array), size, NULL, ZVAL_PTR_DTOR, 0);
```

7.7.6.2 插入、更新元素

数组元素的插入、更新主要有三种情况：key为zend_string、key为普通字符串、key为数值索引，相关的宏及函数：

```
// 1) key为zend_string
```

```
//插入或更新元素，会增加key的refcount
#define zend_hash_update(ht, key, pData) \
        _zend_hash_update(ht, key, pData ZEND_FILE_LINE_CC)

//插入或更新元素，当Bucket类型为indirect时，将pData更新至indirect的值，
而不是更新Bucket
#define zend_hash_update_ind(ht, key, pData) \
        _zend_hash_update_ind(ht, key, pData ZEND_FILE_LINE_CC)

//添加元素，与zend_hash_update()类似，不同的地方在于如果元素已经存在则不
会更新
#define zend_hash_add(ht, key, pData) \
        _zend_hash_add(ht, key, pData ZEND_FILE_LINE_CC)

//直接插入元素，不管key存在与否，如果存在也不覆盖原来元素，而是当做哈希冲突
处理，所有会出现一个数组中key相同的情况，慎用!!!
#define zend_hash_add_new(ht, key, pData) \
        _zend_hash_add_new(ht, key, pData ZEND_FILE_LINE_CC)

// 2) key为普通字符串：char*

//与上面几个对应，这里的key为普通字符串，会自动生成zend_string的key
#define zend_hash_str_update(ht, key, len, pData) \
        _zend_hash_str_update(ht, key, len, pData ZEND_FILE_LINE
_CC)
#define zend_hash_str_update_ind(ht, key, len, pData) \
        _zend_hash_str_update_ind(ht, key, len, pData ZEND_FILE_
LINE_CC)
#define zend_hash_str_add(ht, key, len, pData) \
        _zend_hash_str_add(ht, key, len, pData ZEND_FILE_LINE_CC)

#define zend_hash_str_add_new(ht, key, len, pData) \
        _zend_hash_str_add_new(ht, key, len, pData ZEND_FILE_LIN
E_CC)

// 3) key为数值索引

//插入元素，h为数值
#define zend_hash_index_add(ht, h, pData) \
        _zend_hash_index_add(ht, h, pData ZEND_FILE_LINE_CC)
```

```
//与zend_hash_add_new()类似
#define zend_hash_index_add_new(ht, h, pData) \
        _zend_hash_index_add_new(ht, h, pData ZEND_FILE_LINE_CC)

//更新第h个元素
#define zend_hash_index_update(ht, h, pData) \
        _zend_hash_index_update(ht, h, pData ZEND_FILE_LINE_CC)

//使用自动索引值
#define zend_hash_next_index_insert(ht, pData) \
        _zend_hash_next_index_insert(ht, pData ZEND_FILE_LINE_CC)


#define zend_hash_next_index_insert_new(ht, pData) \
        _zend_hash_next_index_insert_new(ht, pData ZEND_FILE_LIN
E_CC)
```

7.7.6.3 查找元素

```
//根据zend_string key查找数组元素
ZEND_API zval* ZEND_FASTCALL zend_hash_find(const HashTable *ht,
 zend_string *key);

//根据普通字符串key查找元素
ZEND_API zval* ZEND_FASTCALL zend_hash_str_find(const HashTable
*ht, const char *key, size_t len);

//获取数值索引元素
ZEND_API zval* ZEND_FASTCALL zend_hash_index_find(const HashTabl
e *ht, zend_ulong h);

//判断元素是否存在
ZEND_API zend_bool ZEND_FASTCALL zend_hash_exists(const HashTabl
e *ht, zend_string *key);
ZEND_API zend_bool ZEND_FASTCALL zend_hash_str_exists(const Hash
Table *ht, const char *str, size_t len);
ZEND_API zend_bool ZEND_FASTCALL zend_hash_index_exists(const Ha
shTable *ht, zend_ulong h);

//获取数组元素数
#define zend_hash_num_elements(ht) \
    (ht)->nNumOfElements
//与zend_hash_num_elements()类似，会有一些特殊处理
ZEND_API uint32_t zend_array_count(HashTable *ht);
```

7.7.6.4 删除元素

```
//删除key
ZEND_API int ZEND_FASTCALL zend_hash_del(HashTable *ht, zend_string *key);

//与zend_hash_del()类似，不同地方是如果元素类型为indirect则同时销毁indirect的值
ZEND_API int ZEND_FASTCALL zend_hash_del_ind(HashTable *ht, zend_string *key);
ZEND_API int ZEND_FASTCALL zend_hash_str_del(HashTable *ht, const char *key, size_t len);
ZEND_API int ZEND_FASTCALL zend_hash_str_del_ind(HashTable *ht, const char *key, size_t len);
ZEND_API int ZEND_FASTCALL zend_hash_index_del(HashTable *ht, zend_ulong h);
ZEND_API void ZEND_FASTCALL zend_hash_del_bucket(HashTable *ht, Bucket *p);
```

7.7.6.5 遍历

数组遍历类似foreach的用法，在扩展中可以通过如下的方式遍历：

```
zval *val;
ZEND_HASH_FOREACH_VAL(ht, val) {
    ...
} ZEND_HASH_FOREACH_END();
```

遍历过程中会把数组元素赋值给val，除了上面这个宏还有很多其他用于遍历的宏，这里列几个比较常用的：

```
//遍历获取所有的数值索引
#define ZEND_HASH_FOREACH_NUM_KEY(ht, _h) \
    ZEND_HASH_FOREACH(ht, 0); \
    _h = _p->h;

//遍历获取所有的key
#define ZEND_HASH_FOREACH_STR_KEY(ht, _key) \
    ZEND_HASH_FOREACH(ht, 0); \
    _key = _p->key;

//上面两个的聚合
#define ZEND_HASH_FOREACH_KEY(ht, _h, _key) \
    ZEND_HASH_FOREACH(ht, 0); \
    _h = _p->h; \
    _key = _p->key;

//遍历获取数值索引key及value
#define ZEND_HASH_FOREACH_NUM_KEY_VAL(ht, _h, _val) \
    ZEND_HASH_FOREACH(ht, 0); \
    _h = _p->h; \
    _val = _z;

//遍历获取key及value
#define ZEND_HASH_FOREACH_STR_KEY_VAL(ht, _key, _val) \
    ZEND_HASH_FOREACH(ht, 0); \
    _key = _p->key; \
    _val = _z;

#define ZEND_HASH_FOREACH_KEY_VAL(ht, _h, _key, _val) \
    ZEND_HASH_FOREACH(ht, 0); \
    _h = _p->h; \
    _key = _p->key; \
    _val = _z;
```

7.7.6.6 其它操作

```
//合并两个数组，将source合并到target，overwrite为元素冲突时是否覆盖
#define zend_hash_merge(target, source, pCopyConstructor, overwr
ite)                        \
    _zend_hash_merge(target, source, pCopyConstructor, overwrite
 ZEND_FILE_LINE_CC)

//导出数组
ZEND_API HashTable* ZEND_FASTCALL zend_array_dup(HashTable *sour
ce);
```

```
#define zend_hash_sort(ht, compare_func, renumber) \
    zend_hash_sort_ex(ht, zend_sort, compare_func, renumber)
```

数组排序，compare_func为typedef int (*compare_func_t)(const void* , const void *)，需要自己定义比较函数，参数类型为*Bucket*，renumber表示是否更改键值，如果为1则会在排序后重新生成各元素的h。PHP中的sort()、rsort()、ksort()等都是基于这个函数实现的。

7.7.6.7 销毁数组

```
ZEND_API void ZEND_FASTCALL zend_array_destroy(HashTable *ht);
```

7.8 常量

常量的具体实现前面章节已经介绍过，这里不再重复。PHP提供了很多用于常量注册的宏，可以在扩展的 `PHP_MINIT_FUNCTION()` 中定义：

```
//注册NULL常量
#define REGISTER_NULL_CONSTANT(name, flags)  \
    zend_register_null_constant((name), sizeof(name)-1, (flags),
 module_number)

//注册bool常量
#define REGISTER_BOOL_CONSTANT(name, bval, flags)  \
    zend_register_bool_constant((name), sizeof(name)-1, (bval),
(flags), module_number)

//注册整形常量
#define REGISTER_LONG_CONSTANT(name, lval, flags)  \
    zend_register_long_constant((name), sizeof(name)-1, (lval),
(flags), module_number)

//注册浮点型常量
#define REGISTER_DOUBLE_CONSTANT(name, dval, flags)  \
    zend_register_double_constant((name), sizeof(name)-1, (dval)
, (flags), module_number)

//注册字符串常量，str类型为char*
#define REGISTER_STRING_CONSTANT(name, str, flags)  \
    zend_register_string_constant((name), sizeof(name)-1, (str),
 (flags), module_number)

//注册字符串常量，截取指定长度，str类型为char*
#define REGISTER_STRINGL_CONSTANT(name, str, len, flags)  \
    zend_register_stringl_constant((name), sizeof(name)-1, (str)
, (len), (flags), module_number)
```

除了上面这些还有 `REGISTER_NS_XXX` 系列的宏用于带namespace的常量注册，另外如果这些类型不能满足需求，则可以通过 `zend_register_constant(zend_constant *c)` 注册，比如常量类型为数组。

```
PHP_MINIT_FUNCTION(mytest)
{
    ...

    REGISTER_STRING_CONSTANT("MY_CONS_1", "this is a constant",
CONST_CS | CONST_PERSISTENT);
}
```

```
echo MY_CONS_1;
=========[output]=========
this is a constant
```

如果在扩展中需要用到其他扩展或内核定义的常量，则可以通过以下函数获取常量的值：

```
ZEND_API zval *zend_get_constant(zend_string *name);
ZEND_API zval *zend_get_constant_str(const char *name, size_t na
me_len);
```

8.1 概述

什么是命名空间？从广义上来说，命名空间是一种封装事物的方法。在很多地方都可以见到这种抽象概念。例如，在操作系统中目录用来将相关文件分组，对于目录中的文件来说，它就扮演了命名空间的角色。具体举个例子，文件 foo.txt 可以同时在目录/home/greg 和 /home/other 中存在，但在同一个目录中不能存在两个 foo.txt 文件。另外，在目录 /home/greg 外访问 foo.txt 文件时，我们必须将目录名以及目录分隔符放在文件名之前得到 /home/greg/foo.txt。这个原理应用到程序设计领域就是命名空间的概念。(引用自php.net)

命名空间主要用来解决两类问题：

- 用户编写的代码与PHP内部的或第三方的类、函数、常量、接口名字冲突
- 为很长的标识符名称创建一个别名的名称，提高源代码的可读性

PHP命名空间提供了一种将相关的类、函数、常量和接口组合到一起的途径，不同命名空间的类、函数、常量、接口相互隔离不会冲突，注意：PHP命名空间只能隔离类、函数、常量和接口，不包括全局变量。

接下来的两节将介绍下PHP命名空间的内部实现，主要从命名空间的定义及使用两个方面分析。

8.2 命名空间的定义

8.2.1 定义语法

命名空间通过关键字namespace 来声明，如果一个文件中包含命名空间，它必须在其它所有代码之前声明命名空间，除了declare关键字以外，也就是说除declare之外任何代码都不能在namespace之前声明。另外，命名空间并没有文件限制，可以在多个文件中声明同一个命名空间，也可以在同一文件中声明多个命名空间。

```
namespace com\aa;

const MY_CONST = 1234;
function my_func(){ /* ... */ }
class my_class { /* ... */ }
```

另外也可以通过{}将类、函数、常量封装在一个命名空间下：

```
namespace com\aa{
    const MY_CONST = 1234;
    function my_func(){ /* ... */ }
    class my_class { /* ... */ }
}
```

但是同一个文件中这两种定义方式不能混用，下面这样的定义将是非法的：

```
namespace com\aa{
    /* ... */
}

namespace com\bb;
/* ... */
```

如果没有定义任何命名空间，所有的类、函数和常量的定义都是在全局空间，与 PHP 引入命名空间概念前一样。

8.2.2 内部实现

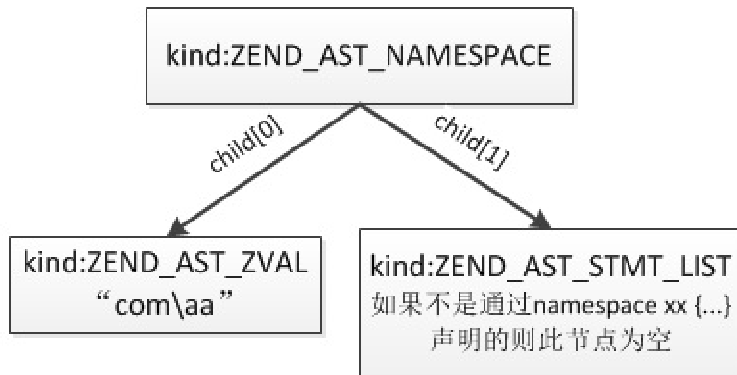
命名空间的实现实际比较简单，当声明了一个命名空间后，接下来编译类、函数和常量时会把类名、函数名和常量名统一加上命名空间的名称作为前缀存储，也就是说声明在命名空间中的类、函数和常量的实际名称是被修改过的，这样来看他们与普通的定义方式是没有区别的，只是这个前缀是内核帮我们自动添加的，例如：

```
//ns_define.php
namespace com\aa;

const MY_CONST = 1234;
function my_func(){ /* ... */ }
class my_class { /* ... */ }
```

最终MY_CONST、my_func、my_class在EG(zend_constants)、EG(function_table)、EG(class_table)中的实际存储名称被修改为：com\aa\MY_CONST、com\aa\my_func、com\aa\my_class。

下面具体看下编译过程，namespace语法被编译为ZEND_AST_NAMESPACE类型的语法树节点，它有两个子节点：child[0]为命名空间的名称、child[1]为通过{}方式定义时包裹的语句。



此节点的编译函数为zend_compile_namespace()：

```
void zend_compile_namespace(zend_ast *ast)
{
    zend_ast *name_ast = ast->child[0];
    zend_ast *stmt_ast = ast->child[1];
    zend_string *name;
    zend_bool with_bracket = stmt_ast != NULL;

    //检查声明方式，不允许{}与非{}混用
    ...

    if (FC(current_namespace)) {
        zend_string_release(FC(current_namespace));
    }

    if (name_ast) {
        name = zend_ast_get_str(name_ast);

        if (ZEND_FETCH_CLASS_DEFAULT != zend_get_class_fetch_typ
e(name)) {
            zend_error_noreturn(E_COMPILE_ERROR, "Cannot use '%s
' as namespace name", ZSTR_VAL(name));
        }
        //将命名空间名称保存到FC(current_namespace)
        FC(current_namespace) = zend_string_copy(name);
    } else {
        FC(current_namespace) = NULL;
    }

    //重置use导入的命名空间符号表
    zend_reset_import_tables();
    ...
    if (stmt_ast) {
        //如果是通过namespace xxx { ... }这种方式声明的则直接编译{}中
的语句
        zend_compile_top_stmt(stmt_ast);
        zend_end_namespace();
    }
}
```

从上面的编译过程可以看出，命名空间定义的编译过程非常简单，最主要的操作是把FC(current_namespace)设置为当前定义的命名空间名称，FC()这个宏为:CG(file_context)，前面曾介绍过，file_context是在编译过程中使用的一个结构：

```
typedef struct _zend_file_context {
    zend_declarables declarables;
    znode implementing_class;

    //当前所属namespace
    zend_string *current_namespace;
    //是否在namespace中
    zend_bool in_namespace;
    //当前namespace是否为{}定义
    zend_bool has_bracketed_namespaces;

    //下面这三个值在后面介绍use时再说明，这里忽略即可
    HashTable *imports;
    HashTable *imports_function;
    HashTable *imports_const;
} zend_file_context;
```

编译完namespace声明语句后接着编译下面的语句，此后定义的类、函数、常量均属于此命名空间，直到遇到下一个namespace的定义，接下来继续分析下这三种类型编译过程中有何不同之处。

(1)编译类、函数

前面章节曾详细介绍过函数、类的编译过程，总结下主要分为两步：第1步是编译函数、类，这个过程将分别生成一条ZEND_DECLARE_FUNCTION、ZEND_DECLARE_CLASS的opcode；第2步是在整个脚本编译的最后执行zend_do_early_binding()，这一步相当于执行ZEND_DECLARE_FUNCTION、ZEND_DECLARE_CLASS，函数、类正是在这一步注册到EG(function_table)、EG(class_table)中去的。

在生成ZEND_DECLARE_FUNCTION、ZEND_DECLARE_CLASS两条opcode时会把函数名、类名的存储位置通过操作数记录下来，然后在zend_do_early_binding()阶段直接获取函数名、类名作为key注册到

EG(function_table)、EG(class_table)中，定义在命名空间中的函数、类的名称修改正是在生成ZEND_DECLARE_FUNCTION、ZEND_DECLARE_CLASS时完成的，下面以函数为例看下具体的处理：

```
//函数的编译方法
void zend_compile_func_decl(znode *result, zend_ast *ast)
{
    ...
    //生成函数声明的opcode：ZEND_DECLARE_FUNCTION
    zend_begin_func_decl(result, op_array, decl);

    //编译参数、函数体
    ...
}
```

```
static void zend_begin_func_decl(znode *result, zend_op_array *o
p_array, zend_ast_decl *decl)
{
    ...
    //获取函数名称
    op_array->function_name = name = zend_prefix_with_ns(unquali
fied_name);
    lcname = zend_string_tolower(name);

    if (FC(imports_function)) {
        //如果通过use导入了其他命名空间则检查函数名称是否已存在
    }
    ....
    //生成一条opcode：ZEND_DECLARE_FUNCTION
    opline = get_next_op(CG(active_op_array));
    opline->opcode = ZEND_DECLARE_FUNCTION;
    //函数名的存储位置记录在op2中
    opline->op2_type = IS_CONST;
    LITERAL_STR(opline->op2, zend_string_copy(lcname));
    ...
}
```

函数名称通过zend_prefix_with_ns()方法获取：

```
zend_string *zend_prefix_with_ns(zend_string *name) {
    if (FC(current_namespace)) {
        //如果当前是在namespace下则拼上namespace名称作为前缀
        zend_string *ns = FC(current_namespace);
        return zend_concat_names(ZSTR_VAL(ns), ZSTR_LEN(ns), ZST
R_VAL(name), ZSTR_LEN(name));
    } else {
        return zend_string_copy(name);
    }
}
```

在zend_prefix_with_ns()方法中如果发现FC(current_namespace)不为空则将函数名加上FC(current_namespace)作为前缀，接下来向EG(function_table)注册时就使用修改后的函数名作为key，类的情况与函数的处理方式相同，不再赘述。

(2)编译常量

常量的编译过程与函数、类基本相同，也是在编译过程获取常量名时检查FC(current_namespace)是否为空，如果不为空表示常量声明在namespace下，则为常量名加上FC(current_namespace)前缀。

总结下命名空间的定义：编译时如果发现定义了一个namespace，则将命名空间名称保存到FC(current_namespace)，编译类、函数、常量时先判断FC(current_namespace)是否为空，如果为空则按正常名称编译，如果不为空则将类名、函数名、常量名加上FC(current_namespace)作为前缀，然后再以修改后的名称注册。整个过程相当于PHP帮我们补全了类名、函数名、常量名。

8.3 命名空间的使用

8.3.1 基本用法

上一节我们知道了定义在命名空间中的类、函数和常量只是加上了namespace名称作为前缀，既然是这样那么在使用时加上同样的前缀是否就可以了呢？答案是肯定的，比如上面那个例子：在com\aa命名空间下定义了一个常量MY_CONST，那么就可以这么使用：

```
include 'ns_define.php';

echo \com\aa\MY_CONST;
```

这种按照实际类名、函数名、常量名使用的方式很容易理解，与普通的类型没有差别，这种以"\"开头使用的名称称之为：完全限定名称，类似于绝对目录的概念，使用这种名称PHP会直接根据"\"之后的名称去对应的符号表中查找(namespace定义时前面是没有加"\"的，所以查找时也会去掉这个字符)。

除了这种形式的名称之外，还有两种形式的名称：

- 非限定名称: 即没有加任何namespace前缀的普通名称，比如my_func()，使用这种名称时如果当前有命名空间则会被解析为：currentnamespace\my_func，如果当前没有命名空间则按照原始名称my_func解析
- 部分限定名称: 即包含namespace前缀，但不是以"\"开始的，比如：aa\my_func()，类似相对路径的概念，这种名称解析规则比较复杂，如果当前空间没有使用use导入任何namespace那么与非限定名称的解析规则相同，即如果当前有命名空间则会把解析为：currentnamespace\aa\my_func，否则解析为aa\my_func，使用use的情况后面再作说明

8.3.2 use导入

使用一个命名空间中的类、函数、常量虽然可以通过完全限定名称的形式访问，但是这种方式需要在每一处使用的地方都加上完整的namespace名称，如果将来namespace名称变更了就需要所有使用的地方都改一遍，这将是很痛苦的一件事，为此，PHP提供了一种命名空间导入/别名的机制，可以通过use关键字将一个命名空间导入或者定义一个别名，然后在使用时就可以通过导入的namespace名称最后一个域或者别名访问，不需要使用完整的名称，比如：

```
//ns_define.php
namespace aa\bb\cc\dd;

const MY_CONST = 1234;
```

可以采用如下几种方式使用：

```
//方式1:
include 'ns_define.php';

use aa\bb\cc\dd;

echo dd\MY_CONST;
```

```
//方式2:
include 'ns_define.php';

use aa\bb\cc;

echo cc\dd\MY_CONST;
```

```
//方式3:
include 'ns_define.php';

use aa\bb\cc\dd as DD;

echo DD\MY_CONST;
```

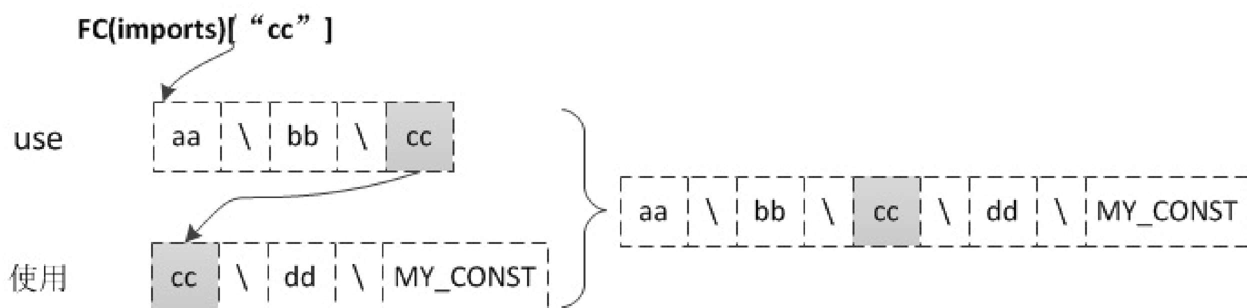
```
//方式4:
include 'ns_define.php';

use aa\bb\cc as CC;

echo CC\dd\MY_CONST;
```

这种机制的实现原理也比较简单：编译期间如果发现use语句 ，那么就将把这个use后的命名空间名称插入一个哈希表：FC(imports)，而哈希表的key就是定义的别名，如果没有定义别名则key使用按"\"分割的最后一节，比如方式2的情况将以cc作为key，即：FC(imports)["cc"] = "aa\bb\cc\dd"；接下来在使用类、函数和常量时会把名称按"\"分割，然后以第一节为key查找FC(imports)，如果找到了则将FC(imports)中保存的名称与使用时的名称拼接在一起，组成完整的名称。实际上这

种机制是把完整的名称切割缩短然后缓存下来，使用时再拼接成完整的名称，也就是内核帮我们组装了名称，对内核而言，最终使用的都是包括完整namespace的名称。



use除了上面介绍的用法外还可以导入一个类，导入后再使用类就不需要加namespace了，例如：

```
//ns_define.php
namespace aa\bb\cc\dd;

class my_class { /* ... */ }
```

```
include 'ns_define.php';
//导入一个类
use aa\bb\cc\dd\my_class;
//直接使用
$obj = new my_class();
var_dump($obj);
```

use的这两种用法实现原理是一样的，都是在编译时通过查找FC(imports)实现的名称补全。从PHP 5.6起，use又提供了两种针对函数、常量的导入，可以通过 `use function xxx` 及 `use const xxx` 导入一个函数、常量，这种用法的实现原理与上面介绍的实际是相同，只是在编译时没有保存到FC(imports)，zend_file_context结构中的另外两个哈希表就是在这种情况下使用的：

```
typedef struct _zend_file_context {
    ...
    //用于保存导入的类或命名空间
    HashTable *imports;
    //用于保存导入的函数
    HashTable *imports_function;
    //用于保存导入的常量
    HashTable *imports_const;
} zend_file_context;
```

简单总结下use的几种不同用法：

- **a.**导入命名空间**:** 导入的名称保存在FC(imports)中，编译使用的语句时搜索此符号表进行补全
- **b.**导入类**:** 导入的名称保存在FC(imports)中，与a不同的是不会根据"\"切割后的最后一节检索，而是直接使用类名查找
- **c.**导入函数**:** 通过 `use function` 导入到FC(imports_function)，补全时先查找FC(imports_function)，如果没有找到则继续按照a的情况处理
- **d.**导入常量**:** 通过 `use const` 导入到FC(imports_const)，补全时先查找FC(imports_const)，如果没有找到则继续按照a的情况处理

```
use aa\bb;                    //导入namespace
use aa\bb\MY_CLASS;           //导入类
use function aa\bb\my_func;  //导入函数
use const aa\bb\MY_CONST;    //导入常量
```

接下来看下内核的具体实现，首先看下use的编译：

```
void zend_compile_use(zend_ast *ast)
{
    zend_string *current_ns = FC(current_namespace);
    //use的类型
    uint32_t type = ast->attr;
    //根据类型获取存储哈希表：FC(imports)、FC(imports_function)、FC(i
mports_const)
    HashTable *current_import = zend_get_import_ht(type);
    ...
```

```
    //use可以同时导入多个
    for (i = 0; i < list->children; ++i) {
        zend_ast *use_ast = list->child[i];
        zend_ast *old_name_ast = use_ast->child[0];
        zend_ast *new_name_ast = use_ast->child[1];
        //old_name为use后的namespace名称，new_name为as定义的别名
        zend_string *old_name = zend_ast_get_str(old_name_ast);
        zend_string *new_name, *lookup_name;

        if (new_name_ast) {
            //如果有as别名则直接使用
            new_name = zend_string_copy(zend_ast_get_str(new_nam
e_ast));
        } else {
            const char *unqualified_name;
            size_t unqualified_name_len;
            if (zend_get_unqualified_name(old_name, &unqualified
_name, &unqualified_name_len)) {
                //按"\"分割，取最后一节为new_name
                new_name = zend_string_init(unqualified_name, un
qualified_name_len, 0);
            } else {
                //名称中没有"\"：use aa
                new_name = zend_string_copy(old_name);
            }
        }
        //如果是use const则大小写敏感，其它用法都转为小写
        if (case_sensitive) {
            lookup_name = zend_string_copy(new_name);
        } else {
            lookup_name = zend_string_tolower(new_name);
        }
        ...
        if (current_ns) {
            //如果当前是在命名空间中则需要检查名称是否冲突
            ...
        }

        //插入FC(imports/imports_function/imports_const)，key为lo
okup_name，value为old_name
```

```
        if (!zend_hash_add_ptr(current_import, lookup_name, old_
name)) {
            ...
        }
    }
}
```

从use的编译过程可以看到，编译时的主要处理是把use导入的名称以别名或最后分节为key存储到对应的哈希表中，接下来我们看下在编译使用类、函数、常量的语句时是如何处理的。使用的语法类型比较多，比如类的使用就有new、访问静态属性、调用静态方法等，但是不管什么语句都会经历获取类名、函数名、常量名这一步，类名的补全就是在这一步完成的。

(1)补全类名

编译时通过zend_resolve_class_name()方法进行类名补全，如果没有任何namespace那么就返回原始的类名，比如编译 `new my_class()` 时，首先会把"my_class"传入该函数，如果查找FC(imports)后发现是一个use导入的类则把补全后的完整名称返回，然后再进行后续的处理。

```
zend_string *zend_resolve_class_name(zend_string *name, uint32_t
 type)
{
    char *compound;
    //"namespace\xxx\类名"这种用法表示使用当前命名空间
    if (type == ZEND_NAME_RELATIVE) {
        return zend_prefix_with_ns(name);
    }

    //完全限定的形式：new \aa\bb\my_class()
    if (type == ZEND_NAME_FQ || ZSTR_VAL(name)[0] == '\\') {
        if (ZSTR_VAL(name)[0] == '\\') {
            name = zend_string_init(ZSTR_VAL(name) + 1, ZSTR_LEN
(name) - 1, 0);
        } else {
            zend_string_addref(name);
        }
        ...
        return name;
```

```
    }

    //如果当前脚本有通过use导入namespace
    if (FC(imports)) {
        compound = memchr(ZSTR_VAL(name), '\\', ZSTR_LEN(name));
        if (compound) {
            // 1) 没有直接导入一个类的情况，用法a
            //名称中包括"\"，比如：new aa\bb\my_class()
            size_t len = compound - ZSTR_VAL(name);
            //根据按"\"分割后的最后一节为key查找FC(imports)
            zend_string *import_name =
                zend_hash_find_ptr_lc(FC(imports), ZSTR_VAL(name
), len);
            //如果找到了表示通过use导入了namespace
            if (import_name) {
                return zend_concat_names(
                    ZSTR_VAL(import_name), ZSTR_LEN(import_name)
, ZSTR_VAL(name) + len + 1, ZSTR_LEN(name) - len - 1);
            }
        } else {
            // 2) 通过use导入一个类的情况，用法b
            //直接根据原始类名查找
            zend_string *import_name
                = zend_hash_find_ptr_lc(FC(imports), ZSTR_VAL(na
me), ZSTR_LEN(name));

            if (import_name) {
                return zend_string_copy(import_name);
            }
        }
    }
    //没有使用use或没命中任何use导入的namespace，按照基本用法处理：如果当
前在一个namespace下则解释为currentnamespace\my_class
    return zend_prefix_with_ns(name);
}
```

此方法除了类的名称后还有一个type参数，这个参数是解析语法是根据使用方式确定的，共有三种类型：

- **ZEND_NAME_NOT_FQ:** 非限定名称，也就是普通的类名，没有加

namespace，比如：new my_class()

- **ZEND_NAME_RELATIVE:** 相对名称，强制按照当前所属命名空间解析，使用时通过在类前加"namespace\xx"，比如：new namespace\my_class()，如果当前是全局空间则等价于:new my_class，如果当前命名空间为currentnamespace，则解析为"currentnamespace\my_class"
- **ZEND_NAME_FQ:** 完全限定名称，即以"\"开头的

(2)补全函数名、常量名

函数与常量名称的补全操作是相同的:

```
//补全函数名称
zend_string *zend_resolve_function_name(zend_string *name, uint3
2_t type, zend_bool *is_fully_qualified)
{
    return zend_resolve_non_class_name(
        name, type, is_fully_qualified, 0, FC(imports_function))
;
}
//补全常量名称
zend_string *zend_resolve_const_name(zend_string *name, uint32_t
 type, zend_bool *is_fully_qualified)
    return zend_resolve_non_class_name(
        name, type, is_fully_qualified, 1, FC(imports_const));
}
```

可以看到函数与常量最终调用同一方法处理，不同点在于传入了各自的存储哈希表：

```
zend_string *zend_resolve_non_class_name(
    zend_string *name, uint32_t type, zend_bool *is_fully_qualif
ied,
    zend_bool case_sensitive, HashTable *current_import_sub
) {
    char *compound;
    *is_fully_qualified = 0;
    //完整名称，直接返回，不需要补全
    if (ZSTR_VAL(name)[0] == '\\') {
        *is_fully_qualified = 1;
        return zend_string_init(ZSTR_VAL(name) + 1, ZSTR_LEN(nam
e) - 1, 0);
    }
    //与类的用法相同
    if (type == ZEND_NAME_RELATIVE) {
        *is_fully_qualified = 1;
        return zend_prefix_with_ns(name);
    }
    //current_import_sub如果是函数则为FC(imports_function)，否则为FC
(imports_const)
    if (current_import_sub) {
        //查找FC(imports_function)或FC(imports_const)
        ...
    }
    //查找FC(imports)
    compound = memchr(ZSTR_VAL(name), '\\', ZSTR_LEN(name));
    ...

    return zend_prefix_with_ns(name);
}
```

可以看到，函数与常量的的补全逻辑只是优先用原始名称去FC(imports_function)或FC(imports_const)查找，如果没有找到再去FC(imports)中匹配。如果我们这样导入了一个函数：`use function aa\bb\my_func;`，编译`my_func()`会在FC(imports_function)中根据"my_func"找到"aa\bb\my_func"，从而使用完整的这个名称。

8.3.3 动态用法

前面介绍的这些命名空间的使用都是名称为CONST类型的情况，所有的处理都是在编译环节完成的，PHP是动态语言，能否动态使用命名空间呢？举个例子：

```
$class_name = "\aa\bb\my_class";
$obj = new $class_name;
```

如果类似这样的用法只能只用完全限定名称，也就是按照实际存储的名称使用，无法进行自动名称补全。

附录1：break/continue按标签中断语法实现

1.1 背景

首先看下目前PHP中break/continue多层循环的情况：

```
//loop1
while(...){
    //loop2
    for(...){
        //loop3
        foreach(...){
            ...
            break 2;
        }

    }
    //loop2 end
    ...

}
```

`break 2` 表示要中断往上数两层也就是loop2这层循环，`break 2` 之后将从loop2 end开始继续执行。PHP的break、continue只能根据数值中断对应的循环，当嵌套循环比较多的时候这种方式维护起来就变得很不方便，需要一层层的去数要中断的循环。

了解Go语言的读者应该知道在Go中可以按照标签中断，举个例子来看：

```
//test.go
func main() {
loop1:
    for i := 0; i < 2; i++ {
        fmt.Println("loop1")

        for j := 0; j < 5; j++ {
            fmt.Println("  loop2")
            if j == 2 {
                break loop1
            }
        }
    }
}
```

`go run test.go` 将输出：

```
loop1
  loop2
  loop2
  loop2
```

`break loop1` 这种语法在PHP中是不支持的，接下来我们就对PHP进行改造，让PHP实现同样的功能。

1.2 实现

想让PHP支持类似Go语言那样的语法首先需要明确PHP中循环及中断语句的实现，关于这两部分内容前面《PHP基础语法实现》一章已经详细介绍过了，这里再简单概括下实现的关键点:

- 不管是哪种循环结构，其编译时都生成了一个 `zend_brk_cont_element` 结构，此结构记录着这个循环break、continue要跳转的位置，以及嵌套的父层循环
- break/continue编译时分为两个步骤：首先初步编译为临时opcode，此opcode记录着break/continue所在循环层以及要中断的层级(即: `break n` ，默认n=1)；然后在脚本全部编译完之后的pass_two()中，根据当前循环层及中断的

层级n向上查找对应的循环层，最后根据查找到的要中断的循环 `zend_brk_cont_element` 结构得到对应的跳转位置，生成一条ZEND_JMP指令

仔细研究循环、中断的实现可以发现，这里面的关键就在于找到break/continue要中断的那层循环，嵌套循环之间是链表的结构，所以目前的查找就变得很容易了，直接从break/continue当前循环层向前移动n即可。

标签在内核中通过HashTable的结构保存(即：CG(context).labels)，key就是标签名，标签会记录当前opcode的位置，我们要实现 `break 标签` 的语法需要根据标签取到循环，因此我们为标签赋予一种新的含义：循环标签，只有标签紧挨着循环的才认为是这种含义，比如：

```
loop1:
for(...){
    ...
}
```

标签与循环之间有其它表达式的则只能认为是普通标签：

```
loop1:
$a = 123;
for(...){
}
```

既然要按照标签进行break、continue，那么很容易想到把中断的循环层级id保存到标签中，编译break/continue时先查找标签，再查找循环的 `zend_brk_cont_element` 即可，这样实现的话需要循环编译时将自己 `zend_brk_cont_element` 的存储位置保存到标签中，标签的结构需要修改，另外一个问题是标签编译不会生成任何opcode，循环结构无法直接根据上一条opcode判断它是不是 循环标签 ，所以我们换一种方式实现，具体思路如下：

- **(1)** 循环结构开始编译前先编译一条空opcode(ZEND_NOP)，用于标识这是一个循环，并把这个循环 `zend_brk_cont_element` 的存储位置记录在此opcode中
- **(2)** break编译时如果发现是一个标签，则从CG(context).labels)中取出标签结构，然后判断此标签的下一条opcode是否为ZEND_NOP，如果不是则说明这

不是一个 **>**循环标签 ，无法break/continue，如果是则取出循环结构

- **(3)** 得到循环结构之后的处理就比较简单了，但是此时还不能直接编译为ZEND_JMP，因为循环可能还未编译完成，break只能编译为临时opcode，这里可以把标签标记的循环存储位置记录在临时opcode中，然后在pass_two()中再重新获取，需要对pass_two()中的逻辑进行改动，为减少改动，这个地方转化一下实现方式：计算label标记的循环相对break所在循环的位置，也就是转为现有的 `break n` ，这样以来就无需对pass_two()进行改动了

接下来看下具体的实现，以for为例。

(1) 编译循环语句

```
void zend_compile_for(zend_ast *ast)
{
    zend_ast *init_ast = ast->child[0];
    zend_ast *cond_ast = ast->child[1];
    zend_ast *loop_ast = ast->child[2];
    zend_ast *stmt_ast = ast->child[3];

    znode result;
    uint32_t opnum_start, opnum_jmp, opnum_loop;
    zend_op *mark_look_opline;

    //新增：创建一条空opcode，用于标识接下来是一个循环结构
    mark_look_opline = zend_emit_op(NULL, ZEND_NOP, NULL, NULL);

    zend_compile_expr_list(&result, init_ast);
    zend_do_free(&result);

    opnum_jmp = zend_emit_jump(0);

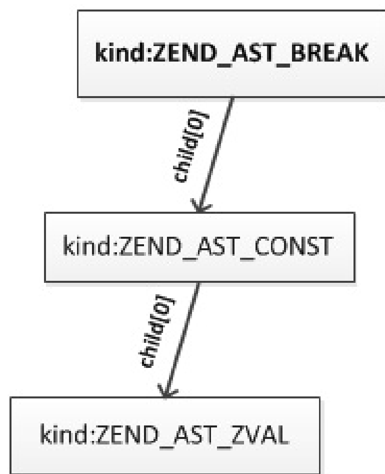
    zend_begin_loop(ZEND_NOP, NULL);

    //新增：保存当前循环的brk，同时为了防止与其它ZEND_NOP混淆，把op1标为-1

    mark_look_opline->op1.var = -1;
    mark_look_opline->extended_value = CG(context).current_brk_c
ont;
    ...
}
```

(2) 编译中断语句

首先明确一点： `break label` 将被编译为以下语法结构：



`ZEND_AST_BREAK` 只有一个子节点，如果是数值那么这个子节点类型为 `ZEND_AST_ZVAL` ，如果是标签则类型是 `ZEND_AST_CONST` ， `ZEND_AST_CONST` 也有一个类型为 `ZEND_AST_ZVAL` 子节点。下面看下break/continue修改后的编译逻辑：

```
void zend_compile_break_continue(zend_ast *ast)
{
    zend_ast *depth_ast = ast->child[0];

    zend_op *opline;
    int depth;

    ZEND_ASSERT(ast->kind == ZEND_AST_BREAK || ast->kind == ZEND
_AST_CONTINUE);

    if (CG(context).current_brk_cont == -1) {
        zend_error_noreturn(E_COMPILE_ERROR, "'%s' not in the 'l
oop' or 'switch' context",
            ast->kind == ZEND_AST_BREAK ? "break" : "continue");
    }

    if (depth_ast) {

        switch(depth_ast->kind){
            case ZEND_AST_ZVAL: //break 数值;
                {
                    zval *depth_zv;
```

```
                    depth_zv = zend_ast_get_zval(depth_ast);
                    if (Z_TYPE_P(depth_zv) != IS_LONG || Z_LVAL_
P(depth_zv) < 1) {
                        zend_error_noreturn(E_COMPILE_ERROR, "'%
s' operator accepts only positive numbers",
                                ast->kind == ZEND_AST_BREAK ? "b
reak" : "continue");
                    }

                    depth = Z_LVAL_P(depth_zv);
                    break;
                }
            case ZEND_AST_CONST://break 标签;
                {
                    //获取label名称
                    zend_string *label = zend_ast_get_str(depth_
ast->child[0]);
                    //根据label获取标记的循环，以及相对break所在循环的
位置
                    depth = zend_loop_get_depth_by_label(label);
                    if(depth > 0){
                        goto SET_OP;
                    }
                    break;
                }
            default:
                zend_error_noreturn(E_COMPILE_ERROR, "'%s' opera
tor with non-constant operand "
                        "is no longer supported", ast->kind == Z
END_AST_BREAK ? "break" : "continue");
        }
    } else {
        depth = 1;
    }

    if (!zend_handle_loops_and_finally_ex(depth)) {
        zend_error_noreturn(E_COMPILE_ERROR, "Cannot '%s' %d lev
el%s",
                ast->kind == ZEND_AST_BREAK ? "break" : "continu
e",
```

```
                  depth, depth == 1 ? "" : "s");
    }

SET_OP:
    opline = zend_emit_op(NULL, ast->kind == ZEND_AST_BREAK ? ZE
ND_BRK : ZEND_CONT, NULL, NULL);
    opline->op1.num = CG(context).current_brk_cont;
    opline->op2.num = depth;
}
```

`zend_loop_get_depth_by_label()` 这个函数用来计算标签标记的循环相对break/continue所在循环的层级：

```
int zend_loop_get_depth_by_label(zend_string *label_name)
{
    zval *label_zv;
    zend_label *label;
    zend_op *next_opline;

    if(UNEXPECTED(CG(context).labels == NULL)){
        zend_error_noreturn(E_COMPILE_ERROR, "can't find label:'
%s' or it not mark a loop", ZSTR_VAL(label_name));
    }

    // 1) 查找label
    label_zv = zend_hash_find(CG(context).labels, label_name);
    if(UNEXPECTED(label_zv == NULL)){
        zend_error_noreturn(E_COMPILE_ERROR, "can't find label:'
%s' or it not mark a loop", ZSTR_VAL(label_name));
    }

    label = (zend_label *)Z_PTR_P(label_zv);

    // 2) 获取label下一条opcode
    next_opline = &(CG(active_op_array)->opcodes[label->opline_n
um]);
    if(UNEXPECTED(next_opline == NULL)){
        zend_error_noreturn(E_COMPILE_ERROR, "can't find label:'
%s' or it not mark a loop", ZSTR_VAL(label_name));
```

```
    }

    int label_brk_offset, curr_brk_offset; //标签标识的循环、break当前所在循环
    int depth = 0; //break当前循环至标签循环的层级
    zend_brk_cont_element *brk_cont_element;

    if(next_opline->opcode == ZEND_NOP && next_opline->op1.var == -1){
        label_brk_offset = next_opline->extended_value;
        curr_brk_offset = CG(context).current_brk_cont;

        brk_cont_element = &(CG(active_op_array)->brk_cont_array[curr_brk_offset]);
        //计算标签标记的循环相对位置
        while(1){
            depth++;

            if(label_brk_offset == curr_brk_offset){
                return depth;
            }

            curr_brk_offset = brk_cont_element->parent;
            if(curr_brk_offset < 0){
                //label标识的不是break所在循环
                zend_error_noreturn(E_COMPILE_ERROR, "can't break/conitnue label:'%s' because it not mark a loop", ZSTR_VAL(label_name));
            }
        }
    }else{
        //label没有标识一个循环
        zend_error_noreturn(E_COMPILE_ERROR, "can't break/conitnue label:'%s' because it not mark a loop", ZSTR_VAL(label_name));
    }

    return -1;
}
```

改动后重新编译PHP，然后测试新的语法是否生效：

```
//test.php

loop1:
for($i = 0; $i < 2; $i++){
    echo "loop1\n";

    for($j = 0; $j < 5; $j++){
        echo "  loop2\n";
        if($j == 2){
            break loop1;
        }
    }
}
```

`php test.php` 输出：

```
loop1
  loop2
  loop2
  loop2
```

其它几个循环结构的改动与for相同，有兴趣的可以自己去尝试下。